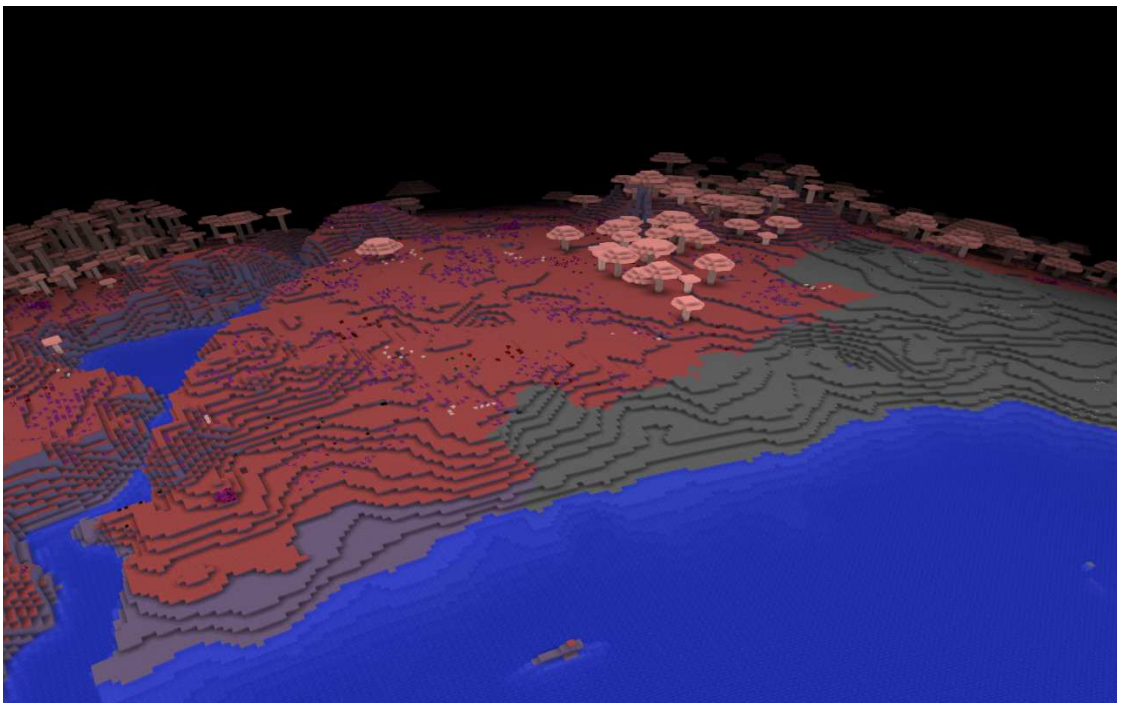
CHALMERS | UNIVERSITY OF GOTHENBURG



Bloxel

Developing a voxel game engine in Java using OpenGL

Bachelor's thesis in Computer Science

ERIC ARNEBÄCK
FELIX BÄRRING
JOHAN HAGE
ANTON LUNDÉN
ANDREAS LÖFMAN
NICLAS OGERYD

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Göteborg, Sweden 2015

Bachelor's thesis 2015:21

Bloxel
Developing a voxel game engine in Java using OpenGL

Eric Arnebäck
Felix Bärring
Johan Hage
Anton Lundén
Andreas Löfman
Niclas Ogeryd

Examiner: Arne Linde
Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg

Cover: Screenshot of the game.
Department of Computer Science and Engineering
Göteborg, Sweden May 2015

Abstract

The purpose of this thesis is to explore the process of creating a voxel game engine, in which features such as procedural terrain generation and world interaction are essential, inspired by pioneering games in the genre such as Minecraft.

The thesis discusses the development process in regards to graphics, physics, and game logic. The game engine is written in Java using the Lightweight Java Game Library for graphics and audio, and employs the Ashley framework to implement an entity-component design pattern.

The project resulted in a simple voxel game, with a procedurally generated environment and basic player-world physics and interaction.

Sammandrag

Syftet med den här avhandlingen är att undersöka skapandet av en voxelspelsmotor, där de viktigaste delarna är procedurell terränggenerering och interaktion med världen, med inspiration från nyskapande spel i genren som Minecraft.

Avhandlingen diskuterar utvecklingsprocessen när det gäller grafik, fysik och spellogik. Spelmotorn är skriven i Java med Lightweight Java Game Library för grafik och ljud, och använder ramverket Ashley för att implementera ett designmönster baserat på enheter och komponenter.

Projektet resulterade i ett enkelt voxelspel, med procedurellt genererad terräng, grundläggande fysik och interaktion mellan spelaren och världen.

Glossary

AO Ambient Occlusion. The lighting phenomenon that the concavities of a scene should be slightly darker than the rest of the scene. 7

CD Abbreviation of Collision Detection. The process of calculating if objects are colliding.. 23

chunk A region consisting of $16 \times 16 \times 112$ blocks.. 10

convex polygon A convex polygon is a polygon where any line drawn through it meets its boundary exactly twice.. 32

gain The amplification of a signal, raising the gain of a sound will result in higher decibel numbers. 41

mesh A 3D object specified from several vertices... 5

NPC Non-player character, any character in a game that is not controlled by a player. 32

panning Gradually moving a sound from one channel to another. 41

pitch frequency of sound. 41

vector Describes a quantity with both magnitude and direction. 22

vertex A point in a 3D space. 5

Contents

1
Introduction

This thesis concerns the creation of a voxel based game engine with the main features being procedural level generation and allowing the player to make changes to the world. In this chapter, we will present the purpose and problem statement of this thesis along with an introduction to the basic concepts of a voxel based game.

1.1 Purpose

The intention of this thesis has been to explore the inner workings of a voxel-based game engine. To achieve this we have developed our own engine. The resulting product should be able to generate a world where the player can move around and interact with their surroundings.

1.2 Background

In recent years, a new genre has emerged, pioneered by games such as Infiniminer [1] and Minecraft [2]. These games let the player explore a world that is mainly made out of cubic blocks. The player can shape the environment in a creative manner by removing and adding blocks to the world, where the number of blocks that can exist appears to be infinite. In Figure 1, a creation made in our game engine can be observed. To be able to have such a huge number of blocks that the user can interact with, an efficient way to store all the required data is needed, and voxels have this property.

A voxel shares similarities with the more well known pixel. The name for pixel is derived from the two words “picture” and “element”. A pixel is essentially a color value in a raster image, that is, a two dimensional array of pixels. Together with the other pixels, they make up a picture that can be displayed on a monitor. Voxels are derived from the word “volume” and “pixel” and differ from pixels in that they exist in three dimensions instead of two. The value of a voxel can be anything that the user wants, and in popular voxel games, they usually represent blocks.

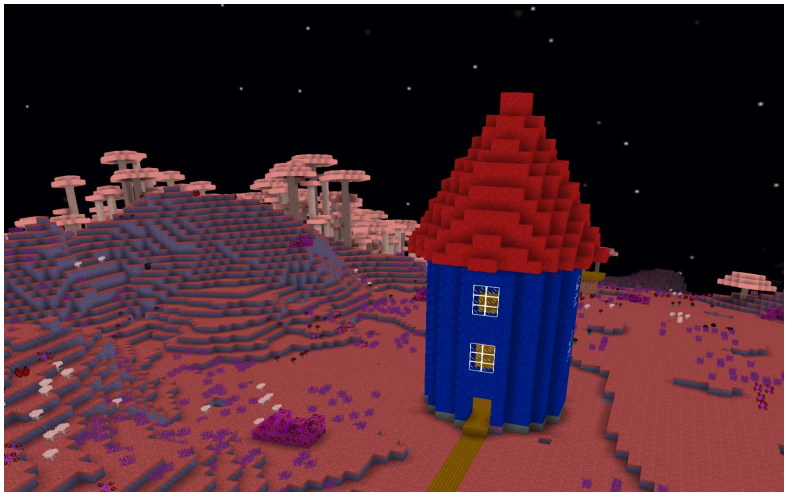


Figure 1: Building made in our engine

1.3 Problem Statement

How does a voxel game engine work, and what is needed to implement one? This is the main question that we intend to explore. In order to acquire an understanding of the techniques used in a voxel game engine, usage of pre-existing tools, such as physics engines or rendering frameworks, has been kept to a minimum. In order to better motivate and test different features of the engine, a simple game which utilizes these has been developed. An engine can have many different components, hence we have limited the scope to the following subsections.

1.3.1 Rendering framework

In a game engine some way of displaying the game world to the player is needed. Since the world primarily consists of voxels, approaches to render a large amount of voxels in an effective way have been explored. Other non-voxel objects in the world are relatively few in comparison, and therefore, having an efficient solution for this is not a high priority. To make the game visually pleasing, various approaches of simulating lighting has been evaluated.

1.3.2 Terrain Generation

One common feature of voxel games is that they have an seemingly infinite and open world, which generally means that the player is left to his own devices to explore the game world [3]. To create an interesting game experience in such a game, the world needs to be varied and rich in features to encourage exploration. Creating such a world manually, which is often the case in other game genres, requires large amount of effort by the creator. The result is often a predefined world that can only be explored once, and hence the replay value can be low. However, in the voxel game genre, procedurally generated terrain is common [2, 4, 5, 6]. This means that every time the player creates a new world, it will be completely unique and different from previous experiences. Various algorithms needed to be implemented in order to create interesting worlds, with features such as height variations, rivers, lakes, and caves.

1.3.3 World Interaction

The player should be able to move around in the world and make changes to it by removing or adding blocks. Some kind of collision detection is needed in order to determine whether or not the player is touching the ground or a wall, and if so, stop it from passing through it. To be able to modify the world, a method is needed that determines which block the player is looking at.

1.3.4 Artificial Intelligence

To create a world with interesting content, non-playable characters (NPC) such as rabbits and cows can be added. To make these NPCs behave as most users would expect, some sort of Artificial intelligence (AI) needs to be utilized. AI is an essential component in many games. Common problems that has to be solved is implementing movement and behaviour, such that the NPC can move and respond to the player in an intelligent manner.

1.3.5 Graphical User Interface

A game can often be customized in many ways to the player's preferences. In order to make this possible in an intuitive manner, a framework for creating graphical user interfaces(GUI) is needed. A GUI needs to be easy to understand while at the same time be feature rich enough so that the user can easily perform the task he or she wants to achieve.

1.3.6 Audio

Audio is a great way to enrich the game experience by providing acoustic feedback of what is happening. For instance, when a user presses a button in a GUI, a sound playing makes it clear that some action has been performed and is handled by the computer. For these reasons a framework that provides an easy way of playing audio is needed.

1.4 Limitations

The main focus of this project has been from a technical standpoint. Creating visually pleasing assets such as textures and models have had a low priority. Advanced rendering algorithms has not been a large concern, partly since there are many other areas that we chose to focus on and also since it might lower the performance of the game.

1.5 Method

In the following sections, our method of work will be described along with the technologies used when implementing this project.

1.5.1 Agile Development

In order to organize the project, we broke it down into cycles using Scrum, which is an iterative agile development methodology [7]. A cycle is a one week period in which each member is assigned one or more tasks that he or she should work on during that period. Unfinished tasks will carry over into the next cycle. The cycle begins with a starting meeting, where the tasks are identified and distributed among the group members. During the week, there are shorter meetings where everyone present their progress and discusses possible problems. Keeping track of what is needed to be done and who was working on what, has been essential, and in our opinion this methodology fit our project well.

1.5.2 Programming Language and Frameworks

We chose to create our engine in Java, mainly because all members have previous experience with the language and its standard libraries. It is also platform independent which allows for each group member to develop in their environment of choice. An additional reason is because our main source of inspiration, Minecraft, was created in Java. Graphics acceleration hardware has become an integral part of real-time rendering. This hardware is specially designed to meet the high performance requirements and can be used with various Graphics APIs. Direct3D and Mantle are two such Application Programming Interfaces(API), but we have chosen to use OpenGL [8], since it has good support on many platforms and also because it is the graphics API that is the most familiar to us. In order to create a window in which we can use OpenGL, some sort of library that provides this is needed. The primary candidates for this task are Java OpenGL (JOGL) and Lightweight Java Game Library (LWJGL) [9]. We choose LWJGL because it also provides an easy way to use sounds and it was also used in our main inspiration Minecraft.

As the number of objects in a game is often high, it can be quite difficult, programming-wise, to keep track of them all. Having a regular object oriented structure quickly creates a huge class tree where changing and rearranging can be difficult [10]. Instead we are using a entity framework called Ashley. A entity framework, or entity-component system, allows for easy implementation of entities and systems handling these entities, without creating large class trees. Ashley was chosen because it is, to the best of our knowledge, the most up-to-date entity framework for Java as compared to other entity frameworks such as Ash and Artemis. In this project, Ashley is used for organized implementation of physics and AI.

1.5.3 Version Control

In order to work effectively in a group, we have used a version control system. There exist many such systems with different qualities, but we chose Git [11] since this is what we are most familiar with and it has worked well in our experience.

2

Rendering Framework

Most games that run on computers need to be able to render its content to a screen in real-time in order to be an enjoyable interactive experience. For the player to feel immersed, the image displayed need to be updated at high speeds, 60 frames per second is considered adequate for most games. It has been demonstrated that a higher frame rate results in the player performing better at the game [12, 13]. Therefore, it is of high importance that the method used for rendering has as high performance as possible, to make sure that frames can be constantly above the 60 frame per second mark. There exists specialized graphics acceleration hardware that make it possible to achieve high performance. To be able to utilize this hardware, there are several APIs, such as OpenGL and DirectX [8, 14].

Furthermore, we wanted to implement lighting in our engine. Lighting in computer graphics means trying to approximate how the light from various light sources, for instance the sun, interacts with a scene. As a result of lighting, some parts of a scene will be darker because another part throws shadows at it, and other parts will be brighter, because some light source is illuminating them. By implementing lighting in a way that is as accurate as possible to lighting in real life, the graphical fidelity of a game can greatly be increased.

In our engine we wanted to implement lighting that is realistic yet cheap. We wanted to implement a lighting engine that supports shadows, torches that illuminate dark areas, and realistic lighting of caves.

2.1 Theory

Rendering of voxels can be done in many ways. Since this project is about making a game engine, only real-time techniques have been considered, which is the topic of this section. We will also describe what approaches there are to implementing lighting in a voxel game.

2.1.1 Rendering of Voxels

There are several ways of rendering voxels, such as ray casting, texture-based volume rendering, splatting and the shear-warp approach [15, 16, 17, 18]. However, to maintain high performance, only polygon rendering of voxels will be used [19, 20], since the graphics hardware that we utilize is specialised for this. The hardware has a pipeline that the programmer can send vertex data through. A vertex is a point in a 3D space. Vertices are used to specify triangles, which are used to build complex meshes that will be processed by the pipeline to generate the final image. In Figure 2 a monkey head mesh built from many triangles can be observed.

(a) Monkey head. (b) Triangulated monkey head.

Figure 2: A monkey head mesh. The images were created in blender

To render a single cube, which is the graphical representation of a voxel in our game, each of the six sides will be represented by two triangles. On these triangles, textures can be mapped in order to create interesting visuals.

2.1.2 Real-time Lighting

There are many advanced real-time lighting models, such as the ones based on so-called BRDFs [21]. However, we will in this section cover only the simplest of all models: the Phong lighting model(Phong).

Phong is a lighting model that uses the direction between a surface and the light sources together with data of the models to compute the shading every frame. Surfaces that are facing the light source will receive *direct light*, that is, will be illuminated directly by the light source. The other surfaces will have their shading estimated by an ambient term. This ambient term is meant to take into account the *indirect light*, which is the light that spreads from other surfaces after receiving light from the light sources. Phong does not consider whether a surface is occluded by another object, as can be observed in Figure 3. To solve this problem, a model for shadows must be implemented in addition to Phong.

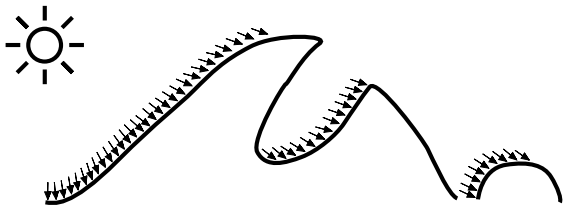


Figure 3: The Phong lighting model.
The arrows indicate where light hits the surface.

2.1.3 Shadows

By adding a model for shadows to Phong, parts of the scene that are occluded from the light source will not be illuminated by direct light, only ambient light. There are two commonly used methods of implementing real-time shadows [22, 23]:

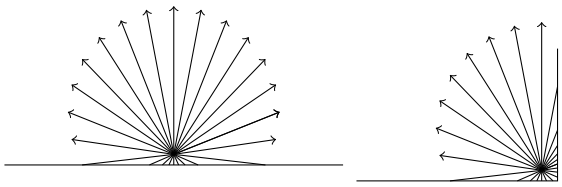
1. Shadow volumes
2. Shadow maps

Shadow volumes requires the creation of many polygons to describe the shadows of every object. This can be prohibitively expensive and also difficult to implement [22].

Shadow maps, while not as accurate as shadow volumes, are both simpler to implement and usually more effective. Shadows maps are implemented by rendering the entire scene from the viewpoint of the light source, and then saving the depth values(the distance from the light source) of the rendered geometry to a texture that is used as a rendering target, called the shadow map [23]. Then the geometry is rendered again, this time from the viewpoint of the actual viewer. In this pass, if the depth value of some geometry is larger than the depth value in the shadow map, it is shadowed, otherwise, it is not.

2.1.4 Ambient Occlusion

Aside from shadows, another point that Phong fails to address is ambient occlusion. Ambient occlusion(AO) attempts to provide a better estimate to the ambient lighting of every point in the scene. If ambient occlusion is implemented correctly, concavities, like for example corners, will be slightly darker than the rest of the scene [24, 25]. AO can easily be implemented by casting equally long rays from every vertex and counting the number of rays that escape. The principle is illustrated in Figure 4. As can be observed, rays are cast in all directions. However, it is impossible to cast in *all* possible directions as they are infinite, thus the number of rays is limited to a finite number. Do note that as the number of cast rays increase, the accuracy of AO improves. For vertices where the number of broken rays is high, the occlusion will be considered high, and therefore the vertex in question will be darkened.



(a) Low AO.

(b) High AO.

Figure 4: AO Raycasting Implementation. Rays without arrow tips are considered broken.

However, the above method requires casting rays from every single vertex every single frame. A less expensive alternative is Screen Space Ambient Occlusion(SSAO), where rays are only cast from every pixel of the screen [26].

Another method of implementing AO was described by M. Lysenko [27]. This is a method that can be used in a voxel game where the voxels are shaped like blocks. To calculate the ambient occlusion of each vertex, we use the information of which blocks are adjacent to it, and by doing so, we can locate all combinations of blocks that form corners. For correct ambient occlusion, these should be darkened. We find that only 8 such combinations are possible, they can be seen in Figure 5. In the fourth row in this image, the vertex is not adjacent to any blocks, thus it is unaffected by ambient occlusion. However, in the first row, we can see the combinations where the vertex forms a corner together with the adjacent blocks, thus it should be darkened. In the second row are the combination were the blocks form less complete corners, thus they should be

less darkened. Furthermore, in the third row, are the combinations of blocks where the corners are even less complete, thus they are even less darkened.

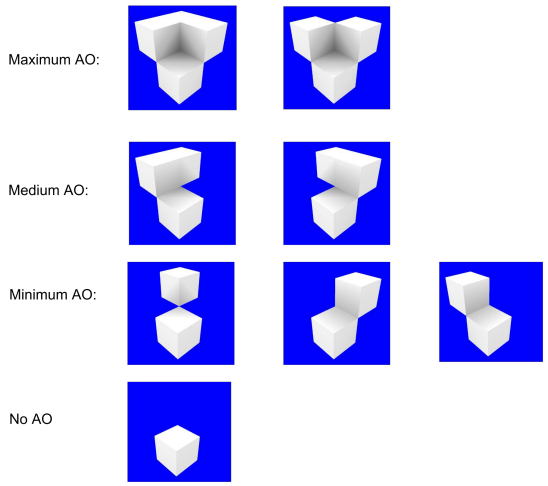


Figure 5: All the 8 combinations of blocks on a vertex. Remade with permission by M. Lysenko [27]

2.1.5 Cellular automata-based lighting model

For games where everything is made out of blocks, there is, aside from Phong, another lighting model that is possible. It was first described by B. Arnold[28], and it is based on the theory of cellular automata.

A *cellular automaton* is a grid of cells where every cell has a finite number of states. This grid of cells is modified by applying preselected rules to the *entire* grid. Every time these rules are applied a new generation is created [29].

Cellular automata can be used to implement lighting in a voxel game [28]. To each block, a light level in the range $[0, 15]$ is assigned. Lighting is modeled with the following rules:

1. Every block that is exposed to the sky should have the maximum light level, which is 15.
2. Solid blocks always have a light level of 0.
3. For every block not exposed to the sky, it holds that the light level is the light level of the neighbour, excluding the diagonal neighbours, that has the greatest light level minus 1.

Note that the above rules makes the assumption that the sun does not have a position. However, the sunlight has a direction, which is always straight down. See Figure 6 for an example of a mushroom from our game that follows the above rules.

15	15	15	15	15	15	15
15	15	15	15	15	15	15
15	0	0	0	0	0	15
15	14	13	0	13	14	15
15	14	13	0	13	14	15
15	14	13	0	13	14	15
0	0	0	0	0	0	0
0	0	0	0	0	0	0

Figure 6: A mushroom that follows the rules of the cellular automata. All blocks are solid, except for those that are white, which are air blocks.

Implementing a naïve algorithm that implements this automaton is not difficult. First, it is made sure that the automata follows rules 1 and 2, meaning that all solid blocks will be given light levels of 0, and all blocks exposed to the sky will be assigned light levels of 15. Following this, rule 3 is applied to every cell in the grid, thus creating a new generation. The algorithm keeps creating new generations in this manner until the grid follows rule 3. To achieve correct lighting that follows rule 3, as many as 16 new generations have to be created.

B. Arnold considered this to be unacceptable, hence he came up with a more efficient approach, which will now be briefly described. First, it is made sure that the automata follows rules 1 and 2. Define *light sources* as blocks with light levels over 0. For every such light source, light is propagated. This means that a breadth-first search is performed to find all blocks who's light levels are two less or more than the current block [30]. To be more specific, for all light sources the following algorithm is performed:

The Light Propagation Algorithm: Let C be the light level of the current cell. We check all the non-diagonal neighbours of that cell. Let N be the light level of such a neighbour. If it holds $N + 1 < C$, then the light level of that neighbour is set to $N - 1$, and we do light propagation for that neighbour as well.

2.2 Result

In this section, we will outline how we implemented the rendering framework of our engine. Our approach of rendering a large amount of voxels with lighting will be explained.

2.2.1 Rendering Large Amounts of Voxels

Our game world consists of large amount of voxels, and rendering them as efficiently as possible is vital for performance. The most naïve approach of rendering would be to make draw calls (call to a function for drawing vertices) for each voxel, except for those that represent air. This would result in very bad utilization of the graphics hardware and the central processing unit (CPU), because of the small amount of vertices that are drawn in each draw call [31]. Our approach to

solving this was to divide the world into chunks of voxels. The size of each chunk is $16 \times 16 \times 112$, where 112 is the height of the world. An image of a chunk can be seen in Figure 7. For each chunk we create a mesh. The vertices of this mesh are generated for the block sides, but only for those sides that are not connected with another solid voxel, in other words, only the visible sides will be rendered. This results in a dramatic reduction of vertices, since it is very common for solid voxels to be connected with other solid voxels.

Moreover, chunks makes it possible to represent our seemingly infinite world in a finite computer. Our world is divided up into chunks, but we are only rendering the chunks that are near the player. The other chunks are saved to the hard disk, and are loaded and rendered only when the player is near them. A chunk is considered near the player if the distance from the chunk to the player is less than or equal to the *render distance*.

Technically the world is not infinite, but has a maximum size of $(2^{32}) * (2^{32})$ chunks, since we use a 32-bit signed integer for storing the x and y coordinates of the chunk. This is however still a very big amount of chunks, and it is highly unlikely that any player will ever move far enough to actually notice this limitation.

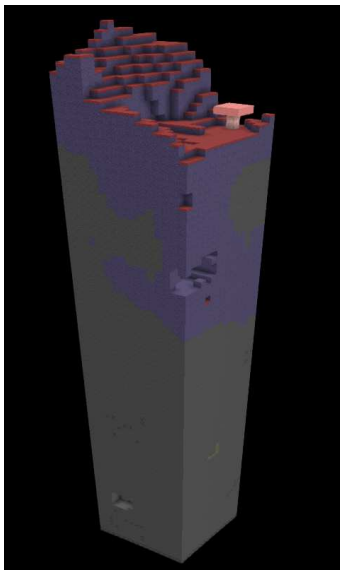


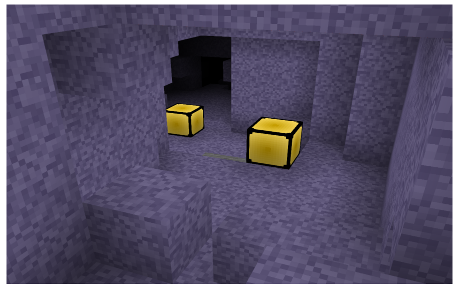
Figure 7: A chunk

2.2.2 Baked Lighting

As will be described in Section 3, caves are generated by our terrain generator. We wanted our lighting to correctly model the lighting of these caves. What most players would expect is that the cave becomes darker as the player progresses deeper into the cave, as can be seen in Figure 8a. This phenomenon can simply be modelled by using the cellular automata approach to lighting, since rule 3 of the cellular automata will ensure that the cave becomes progressively darker, as can be seen in Figure 8b.

We also considered using Phong together with shadow maps or shadow volumes to model this, since it would allow for dynamic shadows that are updated as the sun moves, which is not possible with cellular automata. However, Phong does not at all take occlusion into account,

which means that there is no distinction between caves and non-caves in Phong. In addition, neither shadows maps nor shadows volumes can model cave lighting, because these can only model shadows. Either an object is in shadow, or it is not. They can not model an object being partly in shadow, therefore they cannot model the progressively darker caves either. Therefore, in our game, we choose the cellular automata model of lighting.



(a) The cave lighting we desire.

15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15
15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15
15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15
15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	0	0	0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	0	0	0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	0	0	0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	0	0	0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	0	0	0
15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

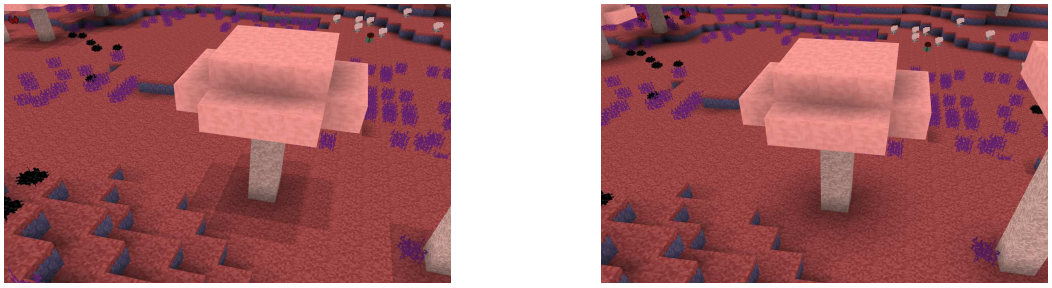
(b) Our desired cave lighting can be modelled using cellular automata..

Figure 8: Images illustrating the kind of lighting we desire in our engine.

For every chunk, we store the light levels of every single block. Whenever a chunk is modified, we redo the light propagation algorithm to update the lighting information. However, since we are performing a fast breadth-first search to update the lighting, this process is unnoticeable to the player. However, an issue with our lighting model is that only the air blocks have been assigned light levels. The air blocks are never rendered, thus it is necessary that the light values of the air blocks affects the brightness of the solid blocks.

In our initial attempt of implementing this, the brightness of a block face was the light level of the air block that is a direct neighbour of that face. Since our lighting model assumes that the sunlight is always shining straight down, it is not necessary to recalculate the lighting every frame, but instead we bake the lighting information into the mesh of every chunk. Only when a chunk is modified, the lighting information is recalculated. The result of this model can be observed in Figure 9a.

As can be observed, the resulting lighting has a blocky appearance, and looks highly unrealistic. Therefore, we tried to find a method of making the lighting have a more smooth appearance. Instead of calculating the lighting for every block face, we attempted calculating the lighting of every separate vertex of the block faces. There are four such vertices for every block face. The lighting value of every vertex is set to the average light level of the eight blocks surrounding every such vertex. The result of this can be seen in Figure 9b.



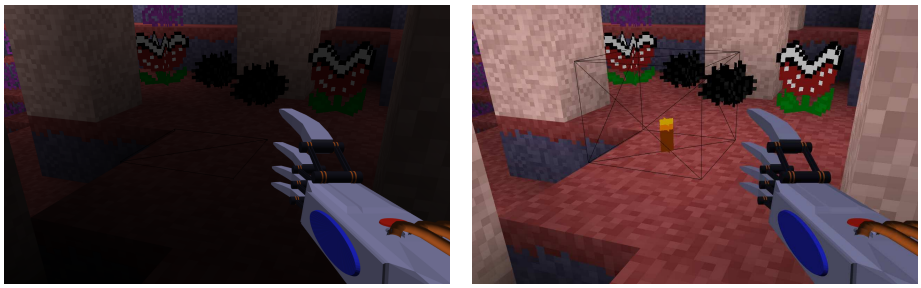
(a) Blocky lighting. (b) Smooth lighting

Figure 9: Blocky lighting and smooth lighting.

2.2.3 Blocklight

In order to implement torches that illuminate dark areas, we decided that every block should contain two kinds of lighting: sunlight and blocklight. Sunlight works just like the cellular automata based lighting we presented in Section 2.1.5. For blocklight on the other hand, all blocks will initially have the blocklight level 0, which is complete darkness. If a torch is placed out, we will simply do light propagation. This light propagation will only affect the blocklight level, and not the sunlight level. If a chunk is modified, the blocklight levels will be recalculated, just like with sunlight.

Finally, to calculate the lighting value of a vertex, we compute the average sunlight level of the surrounding blocks, and in the same manner we calculate the average blocklight level of the surrounding blocks. The sum of these two average values is the final lighting value. In Figure 10, torches are demonstrated.



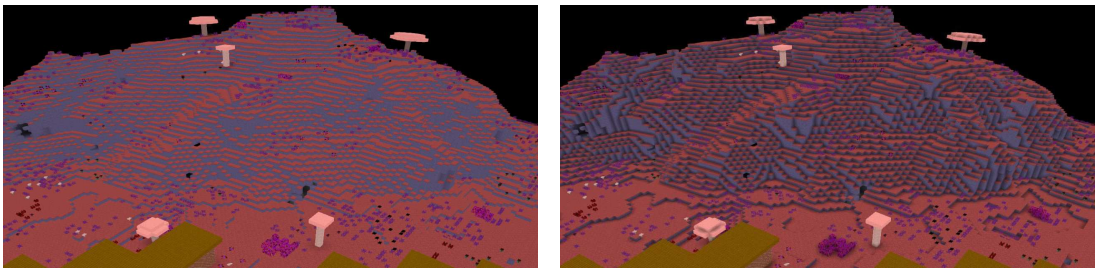
(a) Without torch. (b) With torch.

Figure 10: In the image, most sunlight is blocked by gigantic mushrooms. If a torch is placed out, the scene is illuminated.

2.2.4 Ambient Occlusion

For ambient occlusion, first we implemented SSAO. However, its performance was found to be unacceptable on lower-end graphics cards. Therefore, we implemented the algorithm that was outlined in Figure 5. Just like with sunlight and blocklight, we simply bake this AO information into the mesh of every chunk. The addition of AO greatly improves the appearance of a scene, as

can be seen in Figure 11. Furthermore, thanks to the simplicity of this method, even the lower-end graphics cards were able to maintain a steady frame rate.



(a) Without AO. (b) With AO.

Figure 11: The addition of AO greatly improves the quality of a scene.

2.3 Discussion

In conventional lighting models, such as Phong combined with shadow mapping, the lighting of a scene is computed for every single frame. However, our model that is based on cellular automata, bakes the lighting into the mesh of every chunk, and the lighting only has to be calculated when a chunk is modified. This has the advantage that a powerful graphics card is not necessary to run our game.

Furthermore, our approach has the advantage that it models the lighting of caves in a manner that we think would feel natural to most players. As the player progresses deeper into a cave, the lighting inside the cave should become darker, and this is represented by our light propagation algorithm. Phong and shadow maps do not take occlusion into account, thus they alone can not be used to model this phenomena. However, it could be implemented by performing ambient occlusion by ray casting very long rays.

But our model also has many disadvantages. For example, it assumes that the sun is always in the top of the sky. To implement a moving sun, we would have to redo the light propagation every time the sun moves, which means that the performance becomes on par with Phong again. Furthermore, Phong can be hardware accelerated on the GPU, but our lighting model entirely uses the CPU.

In spite of this, we are content with our lighting model. Because it supports realistic lighting of caves, shadows, and torches, which is exactly what we wanted to implement. Furthermore, the graphical fidelity of the game is greatly improved by the addition of AO.

3

Terrain Generation

As was stated in Section 1.3.2, terrain generation is an important part of a voxel game. In this chapter, the implementation of our terrain generator will be explained. We will first outline the theory behind the techniques we used, and then how these techniques were applied in our game will be outlined. Lastly, pros and cons of our solution will be discussed.

3.1 Theory

A common method of generating terrain is to use a *height map* [19, 32, 33, 34]. A height map is a 2D grid of numbers where every point in the grid is assigned a height. By using a grid where the number of points is small to the degree that the grid is unnoticeable, structures such as mountains and valleys can be described. A height map is commonly generated by using a *noise function*. Such a function produces a pseudorandom output value for every input point (x, y), if the noise is two-dimensional. A noise can be used to calculate a height for every point in a height map.

In this section, different kinds of noise functions will described. Following this, we will outline what techniques can be used for procedurally generating caves. Finally, we will evaluate how biomes can be created.

3.1.1 Noise Algorithms

Noise algorithms can be divided up into two classes: lattice-based noise and point-based nose. In this section, these two classes will briefly be described, and their advantages and disadvantages will be mentioned.

Lattice-Based Noise

The simplest approach to generating *lattice-based noise* is *value noise* [33]. First a pseudorandom number is assigned to every point that is an integer coordinate, the *lattice points*. These points form a grid, also called an *integer lattice*.

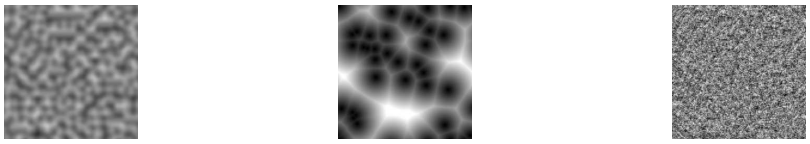
A coordinate defined using integers can easily be used to sample a point by retrieving it from the lattice. However a non-integer coordinate falls between the lattice points and will have to be interpolated between nearby points. There are many interpolation methods available, but two common ones are linear and cubic interpolation.

While value noise is simple to implement, it lacks in visual quality. To solve this problem, K. Perlin invented Perlin noise [35, 36]. In Perlin noise, pseudorandom gradient vectors, in other words unit vectors, are assigned to every lattice point. Perlin noise can be seen in Figure 12a.

Ken Perlin later also developed Simplex noise, which instead of a lattice uses a slight variation called a *simplex grid*. It is an improvement over Perlin noise in that it produces less noticeable arti-

facts [37]. However, the algorithm is patented for dimensions of 3 and higher [38]. An alternative is OpenSimplex Noise, which implements Simplex noise without violating the patent [39].

The noises that were described in this section are all based on some kind of lattice. However, it is not required that this lattice in stored memory, since a hash can be produced from the lattice points, which is then used to look up into a table of pseudorandom numbers. In other words, in order to sample the point (x, y), there is no need to sample any neighbouring points in advance, which makes lattice-based noises very simple to integrate in an application.



(a) Perlin Noise. (b) Worley Noise. (c) Fractional Brownian Motion.

Figure 12: Different noise types.

Point-Based Noise

Another class of noise is point-based noise, and the most well known example of such a noise is Worley noise, which is shown in Figure 12b. To implement Worley noise, first a number of points are randomly distributed. The distance to the closest of these distributed points is used to evaluate the Worley noise at a certain point [40]. But this would require that these points are somehow stored in memory, a property that makes it slightly difficult to integrate Worley noise in an application.

Worley noise can be combined with lattice-based noise to create complex terrain, as has already been demonstrated by K. Musgrave [41].

3.1.2 Fractional Brownian Motion

By computing the sum of several noises of different amplitudes(heights) and frequencies, a fractal called fractional Brownian motion (fBm) is created. Using fBm instead of plain noise often results more visually interesting terrain, and for this reason fBm is often used instead of plain noise [33]. An image of fBm created by Perlin noise can be seen in Figure 12c.

3.1.3 Cave Generation

Several methods of procedurally generating caves have been attempted in the past [32, 42, 43].

Cellular automata can be used to generate caves by continually running a cellular automaton on a 2D grid of randomly generated numbers [42]. The caves created by this approach can be seen in Figure 13. In the figure, white areas are air, and gray areas are rock. This approach can only generate two-dimensional caves, but it can certainly be extended to 3D caves. However, in the figure, it can be observed that there are gray sections that are not connected to the rest of the cave. If this method were to be extended to three dimensions, these gray sections would become floating rocks.

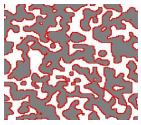


Figure 13: Caves generated by cellular automata [42].

Another approach to generating caves is L-systems, an approach that was attempted by B. Mark et al [32]. An L-system consists of (1)an alphabet of symbols that are used to create strings of symbols, (2)a number of productions rules that define how to expand the symbols into longer strings, and (3)an axiom that defines the initial state of the L-system. An example of an L-system is shown below:

```
1.symbols: A, B
2.productions rules: A->AB, B->A
3.axiom: A
```

The initial state of the system is `A`, the axiom. By applying the production rules to the axiom, the system will reach its second state, `AB`. By again applying the production rules, it will reach its third state, `ABA`. By repeatedly applying the production rules in this manner, a long string of symbols will be created. By interpreting the symbols of this string as drawing commands, a complex fractal structure can be drawn. L-systems are commonly used to describe the structure of objects with a fractal structure, such as certain plants. They can also be used in procedural generation. For instance, P. Müller et al. were able to procedurally generate entire cities through L-systems [44].

B. Mark et al. used L-systems to generate complex cave systems. They defined an L-system for describing caves, expanded it according to the production rules, and then used the symbols of the resulting string to describe the structure of a cave system. The symbols controlled properties such as the branching, the direction and the size of the caves.

Yet another possible approach to cave generation is Perlin worms, where caves are created by having a worm dig through the underground of a height map [43, 32]. Each worm is divided up into segments, and the angle between each segment is decided by a Perlin noise. It is also certainly possible to use a pseudorandom number generator instead of Perlin noise. One of the fastest pseudorandom number generators, and therefore well suited for real-time terrain generation, is the xorshift generator [45].

3.1.4 Biomes

Biomes are a scheme of dividing the Earth into different areas. By dividing procedurally generated terrain into biomes, more variation can be added. In every biome, only certain flora grow, and there are general characteristics for the plant formations of every biome [46]. A simple way of modelling the distribution of biomes is Whittaker's biome classification scheme [47]. In this scheme, biomes of regions are decided by the average temperature and humidity of areas of land, as can be seen in Figure 14.

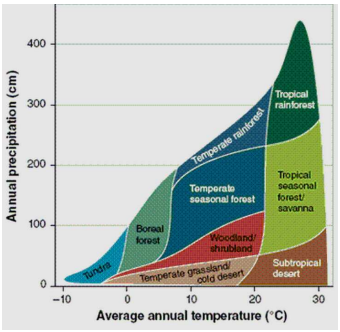


Figure 14: Whittaker's biome classification scheme. By Multichill[1]

There are other schemes for classifying biomes; in the Holdridge scheme, the biome is decided by 3 factors: precipitation, biotemperature and potential evapotranspiration ratio [48]. There are also more advanced, and accurate schemes, such as the Bailey system, in which the Earth is divided up into seven *domains* based on climate, and the domains are further divided based on other climate characteristics [49]. A similar system is the WWF System, where fourteen biomes are identified, and these biomes are further divided into *ecoregions* [50].

3.2 Result

Next we will describe how we implemented a terrain generator based on the techniques outlined in the previous section.

3.2.1 Terrain Generation

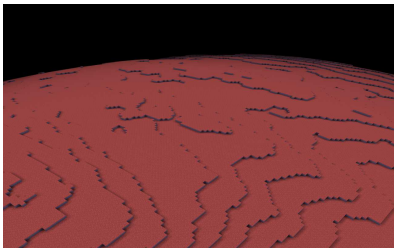
We decided that we would not use Worley noise in our terrain generator, because it noise requires that randomly distributed points are somehow stored in memory, which would increase the complexity in the implementation of the terrain generator.

Therefore, we opted to using only lattice-based noise. It was also decided that fBm would be used instead of plain noise, for the purpose of creating more visually interesting terrain. For generating fBm, we had the choice of using either OpenSimplex or Perlin noise, since value noise lacks in visual quality, and because Simplex noise is patented. It was found that OpenSimplex was twice as slow as Perlin noise, but produced less visible artifacts. So we decided that we would let the user decide which noise type to use for the terrain generation.

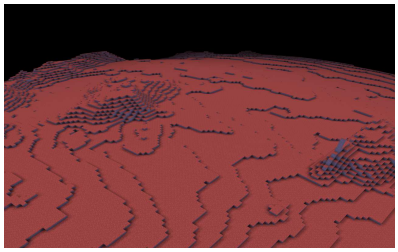
We use fBm to sample the height of each column of blocks, thus creating a height map. Chunk by chunk, we compute this height map. Since we wish to generate an infinite world, generating the entire height map at once is not an option. To solve this problem, the world is generated on demand by generating only the chunks that are within the render distance. Since lattice-based noise requires no knowledge about the neighbouring points, and therefore neither the neighbouring chunks, the height map can trivially be calculated on a chunk-by-chunk basis.

Our height map is created by computing the sum of several fBm-noises. We shall now describe them one by one.

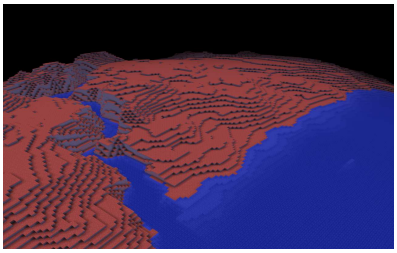
[1] `http://en.wikipedia.org/wiki/File:PrecipitationTempBiomes.jpg`



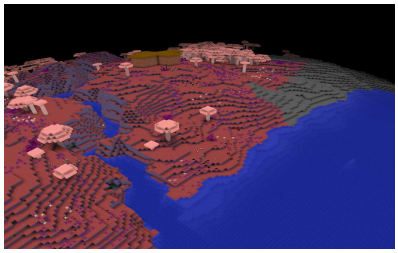
(a) The base noise.



(b) The sum of the hills and base noises.



(c) The sum of the hills, base, ocean, and river noises.



(d) The final terrain. A plains, mushroom forest and wastelands biome can be seen in the image.

Figure 15: The terrain is built by summing several noises.

Base noise: A 2D fBm-noise that creates simple terrain whose height varies slowly. This creates the flat plains of our world. It can be seen in Figure 15a.

Hills Noise: Responsible for creating occasional hills and mountains. Note that this is a 3D noise, and it is the only 3D noise we are using. Instead of sampling a height map value for every column of blocks, we sample a density value for every single block in the column. If this density exceeds a certain value, the block is solid, otherwise, it is air.

We used 3D Perlin noises for the hills noise because plain 2D noise can not produce overhangs. Overhangs are the parts of the terrain that are protruding, and we believed that these were important for creating a visually interesting terrain, and adding more variance.

However, 3D noises were slow to evaluate, and made it difficult to maintain a steady frame rate. We solved this problem by computing the density for only every fourth block, and performing a linear interpolation to calculate the density of the remaining blocks. The sum of the hills and base noises can be seen in Figure 15b.

Ocean Noise: Carves large holes into the terrain, and fills them with water.

River Noise: Carves rivers into the terrain.

The sum of the base, hills, ocean and river noises can be seen in Figure 15c.

Temperature and Humidity Noise: In Section 3.1.4, we outlined several schemes for describing the distribution of biomes. We found that we did not have time to attempt implementing the more complex schemes, thus we decided to only implement the most simple of them all, which is Whittaker's scheme. Furthermore, since this scheme is very simple, the complexity of the terrain generator could, again, be kept to a minimum.

We use two 2D fBm-noises to calculate the humidity and temperature of every block-column in order to decide the biome. To determine the distribution of biomes, inspired by Whittaker's

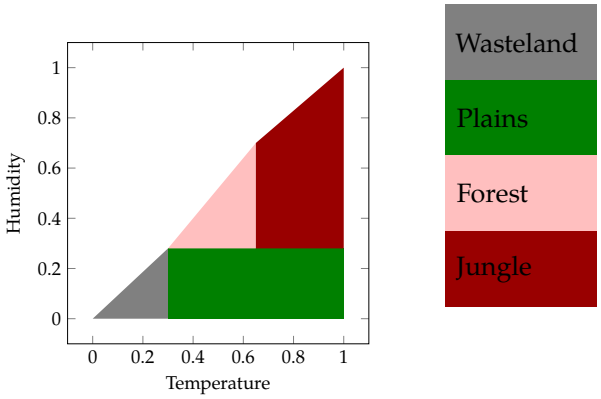


Figure 16: Our biome classifications scheme.

original scheme, we created our own scheme, as can be seen in Figure 16.

3.2.2 Feature Generation

Once the heightmap for a chunk has been generated, *features* are added. Features are structures that are added to every chunk after the heightmap has been generated. The features are:

- Small plants that fit in one single block
- Larger plants that span several blocks
- Resource ores, such as gold ores
- Caves
- Simple houses

Because some features, such as houses, could sometimes span over several chunks, we make sure that all the near chunks of the current chunk have also been generated, and then add features.

Every block column in a chunk has a biome, and the biome affects the column in 3 ways: first, the biome decides the ground block; second, what kind of features can be can be generated in the chunk; third, the average height of the hills noise. If the height is high, then a high valued constant is multiplied with the hills noise, meaning that the biome in question has many high mountains. Our biomes are summarized in Table 1.

There are also features that are created in every biome, which in our case are only resource ores of coal, gold, emerald, ruby, and diamonds. Once biomes and features have been generated, the terrain generation is done. The resulting terrain can be seen in Figure 15d.

3.2.3 Cave Generation

Generating most of the features, such as mushroom forests, plants and resource ores, did not require very complex algorithms, so we will not cover them in any detail here. However, the generation of caves did involve a more complex method: we let three-dimensional worms dig

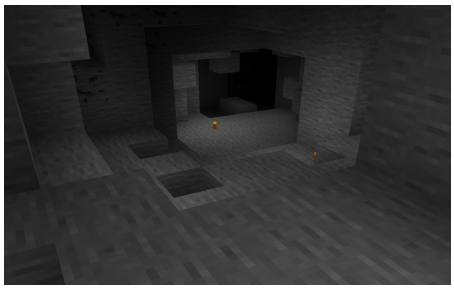
Biome Type	Ground Block	Features	Height
Plains	Red Grass	One-block flora such as grass and flowers, bushes that span several blocks. Medium height mushrooms are sparsely generated, and houses are also generated	Low
Mushroom Forest	Red Grass	One-block flora such as grass and flowers, bushes that span several blocks. Dense medium height mushroom clusters are also generated.	Medium
Mushroom Jungle	Red Grass	One-block flora such as grass and flowers, bushes that span several blocks. Dense high mushroom clusters are also generated.	High
Wasteland	Gray stone	Piles of bones and skulls.	Low

Table 1: Biome information.

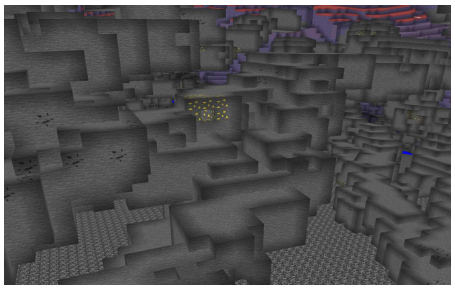
through the underground of the world. Every time we generate a chunk, at a random chance a worm will spawn in that chunk, and start digging in some random direction, replacing all blocks that come in its way with air blocks. Every worm is divided up into segments, and the angle between each such segment is decided by pseudorandom numbers based on a xorshift generator.

To make the caves more visually interesting, we add more randomness by scattering blocks randomly in the caves, and we also let the height of the segments randomly vary, and the angle between each segment is slightly varied by using pseudorandom numbers. Furthermore, the worm completely changes its digging direction at random times, in order to ensure that the caves are not just one-direction tunnels. The inside of a cave can be seen in Figure 17a.

If many worms are spawned, as a result of worms crossing each other, complex cave networks are created. Such a network can be seen in Figure 17b.



(a) The inside of a cave.



(b) A cave network, seen from outside.

Figure 17: Caves from our game

3.3 Discussion

We have in our terrain generator tried approximating the features of the real world as much as possible. By adding together multiple noises, we could represent that the real world does not look similar everywhere. The real world is varied in that it has mountainous areas, plains, and, biomes and so on. This is also true for our terrain generator.

We also thought that Perlin noises were convenient, since they can easily be combined by simply adding them together. This makes it easy to add more features to the terrain. For instance, to implement both rivers and oceans, all we had to do was adding two more Perlin noises.

However, a major issue with Perlin noises is their unpredictability. In a majority of cases, the humidity and temperature Perlin noises result in biomes that are not too big, but not too small either. However, in rare cases, there appears biomes that are only about 1 block big, which looks unnatural. No matter how much we modified the parameters of the Perlin noises, we could not fix this issue.

Another problem with Perlin noises is that often the terrain created by Perlin noises looks the same. For instance, many hills look like each other, and there are very few distinguishing features in them. This is bad from a playability perspective, because if everything looks the same, there are no clear landmarks the player can use when navigating the world.

For generating caves, we opted to using worms, where the angles between the segments are based on pseudorandom numbers. However, as we previously stated, caves can be generated by other method as well. Since cellular automata likely would result in floating rocks, we never attempted this method. L-systems can be used to create complex cave networks. However, implementing them is time consuming, since it would mean that we would have to implement a parser for L-system grammars. In addition, as we illustrated before, our approach can result in relatively complex cave networks, thus we did not feel the need to implement caves using L-systems.

In our case we used pseudorandom numbers to control the angles between the worm segments. Had we used real Perlin worms, we would have controlled the angles with a Perlin noise. However, when comparing the two approaches, we found that they resulted in very similar caves. Because Perlin noises are more complex and therefore take longer time to compute, we decided to use worms based on a xorshift generator.

If we had had more time, we would have liked to add more noise types to the terrain generator, like for example Worley Noise, and test if that would have resulted in more visually interesting terrain.

4

World Interaction

Every game that has some sort of open world where the gameplay takes place needs some sort of interaction in order to be playable. Features like player movement, collision detection, being able to modify the world, and other interaction mechanics are essential for a game to be exciting. This chapter covers what options there are when implementing features like physics, targeting, and other interactive components. The choices of techniques are also motivated and the results of these choices are discussed.

4.1 Theory

This section explains the theory of implementing the features physics, collision detection, and block selection. While collision detection is considered a part of physics, it is explained in its own subsection because of its complexity.

4.1.1 Physics

Movement is an essential part in a dynamic and interactive game. Being able to move as a player gives the feeling of participation and that what one does makes a difference. Having moving objects, in general, makes the game feel more realistic, interesting, and not as stale as it would if everything was static. There are different approaches to implementing movement, such as through predefined or procedural animations [51] or by simply rendering an object at different locations. The most common technique to achieve close to realistic movement is using some sort of physics simulation. Many different physics engines have been made for the purpose of simulating physics in real time. Most of them are used for game development, but there are other uses for them as well, such as in movies and computer graphics [52].There are physics engines that are not open to the public, such as Havoc, PhysX and Vortex as well as physics engines that are free and open-source, such as Bullet [53], Newton Game Dynamics and OpenTissue [54].

Gravity

Simulating gravity is a very basic part of most physics engines since without it actions such as jumping and falling would not feel as realistic. Modifying the strength of the gravitational pull can give the feeling of being on a different planet where things fall faster or slower because of the differently sized planet. Removing the gravity feature would result in a space-like environment where an object would keep its velocity at all time unless it collides with another object.

Implementing gravity can be done differently depending on the design of the physics engine. If the physics engine is built around acceleration, meaning that all objects keeps track of a velocity vector, then gravity would simply mean adding a velocity pointing straight down. In modern games, this is not as common, as physics engines today are often much bigger and are programmed to be more general, in other words, they can handle many different scenarios with

close-to-realism precision. These physics engines simulate rigid bodies [55], which are bodies that keep their shape no matter the forces applied to them [56]. Gravity, in this case, is implemented by simulating a force being applied to the objects. This is closer to how it works in reality but, while easy to implement, requires implementation of rigid bodies.

Player Movement

As mentioned in Section 4.1.1, movement is very important for a game to feel dynamic. Movement has previously been implemented in numerous different ways, mainly because it often differs a lot between [57] games [58]. Because of these vast amount of methods of implementing movement, this section will only explain the most basic ones that could fit this project.

With modern physics engines mainly using full rigid-body physics, movement is achieved by simulating either forces or impulses being applied to these bodies. From a player movement perspective, this would mean applying forces to the player as a response to input. While this yields a more realistic result, an easier option would be to increase the velocity in the direction of a pressed directional key. Doing it this way requires some sort of method to limit the velocity as well as a mechanism to decrease the velocity when the player is supposed to slow down naturally. A third and even simpler option would be to move the player at a fixed speed while a directional key is held. While this way makes for easy implementation, it can make the movement feel very static as accelerating and stopping happens instantly.

4.1.2 Collision Detection

In order to implement realistic physics, collision needs to be resolved. This is done with the part of a physics engine called collision detection (CD). CD is what keeps track of when objects are intersecting, or colliding with, each other. After CD has been performed, the collision response is done, which moves objects based on the data acquired from the CD. It is therefore vital that the CD of a game is implemented with a close to no chance of failure, since failure can lead to the player getting stuck or falling through the world. According to buildnewgames.com [59], CD is the most difficult part of physics to implement, mainly because it is a very expensive operation which therefore needs to be as optimized as possible.

Solely using CD can often become problematic since pairwise testing, testing every object against each other every frame, quickly becomes too costly as the number of objects increases. This problem can be solved by dividing CD into two phases: the broad phase and the narrow phase. The broad phase is when the physics engine evaluates what objects are likely to collide during a frame. Objects are grouped so that moving objects are in the same group as the objects they are likely to collide with. This drastically reduces the amount of pairs that are checked for collision during the narrow phase [60].

Pairwise testing results in a complexity of $O(n^2)$, n being the number of objects, which are all being tested against each other. The broad phase can be implemented using either what is called a sweep-and-prune algorithm or spatial partitioning. This results in a complexity of $O(n \log n)$, depending on the implementation and situation [25].

Spatial partitioning is to divide space into sections, in order to easily be able to locate certain objects or parts of an object. Spatial partitioning can be done with techniques such as quadtrees, octrees, Binary Space Partitioning trees (BSP-tree), R-trees or spatial grids [61]. It can be used for other things than CD, such as culling. Figure 18 shows an example of an octree. After the spatial

partitioning has been done, the narrow phase calculates CD between objects within the same section of the partitioning.

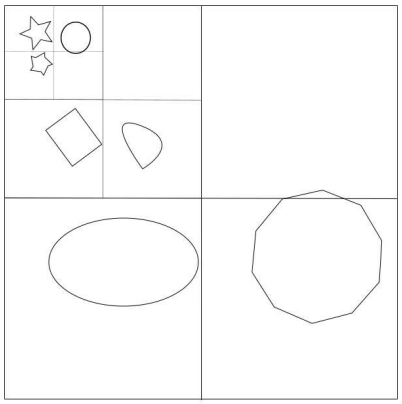


Figure 18: Space partitioned using an octree

Sweep-and-prune is similar to spatial partitioning in that it divides the space into sections, however, with sweep-and-prune it is done one axis at a time. The sweep-and-prune algorithm stores the start and end of each object in a sorted list, with one list for each axis. An object that starts before another object has ended overlaps that other object on the specified axis. Objects overlapping on every axis are likely to collide. This makes the sweep-and-prune algorithm very fast at evaluating what objects are close to each other, especially if not many objects are moving at the same time, since that means that the lists do not have to be re-sorted.

There are many different approaches to implementing CD, each one performing better for different situations. What is common between the different methods of implementing CD is that they all need some sort of geometrical representation of all of the objects that should be able to collide. Two common methods of representing objects are bounding volumes and ray casting.

Ray casting, in short, means casting lines, or rays, from certain points at an object. These rays are then tested for intersection against nearby objects. If a ray is intersecting an object, the object from which the ray was cast is colliding. The response of the collision depends on where the ray intersects the object. Figure 19 shows an example of this. This is a very approximate method of doing CD, but can yield a very efficient result in certain situations [25], especially so since ray casting is an analytic test which is very fast.

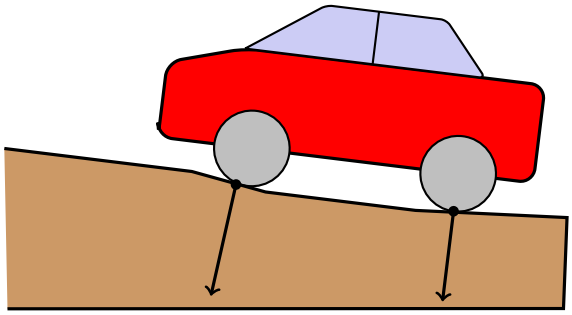


Figure 19: A car with two rays cast from the wheels to detect whether the wheels are in contact with the ground

Bounding volumes, being the more commonly used method of representing objects [62], is a way of surrounding an object with a shape that is used when calculating collision. The more similar the shape of the bounding box is to the objects it is representing, the more realistic collisions will be. These are five different, commonly used bounding volumes: Sphere, Axis-Aligned Bounding Box (AABB), Oriented Bounding Box (OBB), k-direction Discrete Orientation Polytope (k-DOP, k being the number of directions used to form the bounding volume) and convex hull [63]. Figure 20 shows examples of these. Each of these has different methods of calculating intersection. This means that using different bounding volumes requires further implementation. Each of the bounding volumes are different in efficiency, Sphere being the most efficient and convex hull being the most expensive, but they also fit various objects differently well. For example, while a convex hull fits most objects pretty well, a round object would be approximated better, and much more efficiently by a Sphere bounding volume.

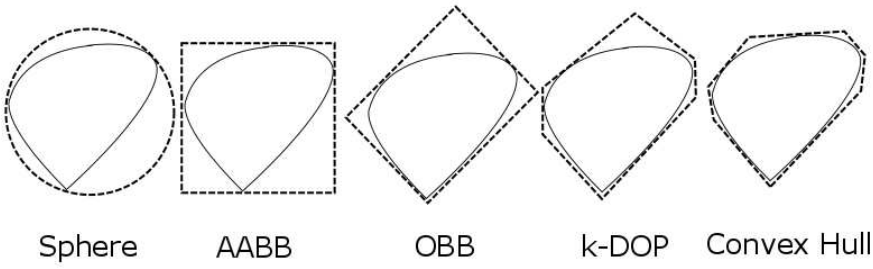


Figure 20: A shape surrounded by five different kinds of bounding volumes

4.1.3 Block Selection

A recurring theme that is common in most voxel games is the ability to interact with the world, not only in the sense of being able to move around, but also the ability for the player to change and shape it as they see fit. This is commonly achieved by placing and removing blocks, which requires the ability to have the player select a voxel in the world for modification.

There are several methods which can be utilized in order to determine the block at which the player is looking at. One solution for block selection is to use ray casting, see Section 4.1.2 for a description of this method. Unfortunately this method can become quite expensive if it is used naively, without doing any spatial partitioning of the game world.

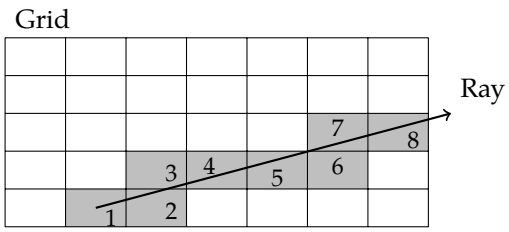


Figure 21: 2D illustration of the grid traversal algorithm

A solution to this problem is to partition the objects to be intersected, into a voxel grid. An algorithm developed for ray tracing a voxel-based grid was first introduced by Andrew Woo and John Amanatides in 1987 [64]. Figure 21 shows a 2D example of the algorithm where the objective of the algorithm is to visit the voxels in numerically increasing order.

The algorithm works by first identifying a starting point at the origin of the ray and saving the direction of the ray which gives the direction of iteration. After that it is a process of looping through the following steps until a solid voxel is found.

1. For each axis, save the distance required to pass the border into the next voxel.
2. Evaluate these values and move in the direction of the axis with the lowest distance needed
3. Check the status of the voxel at the current position, if it is non-empty terminate the algorithm and return it.
4. Repeat until the end of the grid is reached

4.2 Result

This section will give an in depth description of the different techniques that were used for implementing the different world interaction features. It will also cover how and why these techniques were used, followed by how it ended up working out for the project.

4.2.1 Physics

Even though there are several physics engines already made, we opted to implement a custom physics engine. While most of the open-source physics engines are very general, realistic, and well developed, there are several reasons not to use any of them and instead create a custom physics engine, the main reasons being control and speed [65]. Making a physics engine that is not very general but fits a certain project well will most likely result in a faster engine.

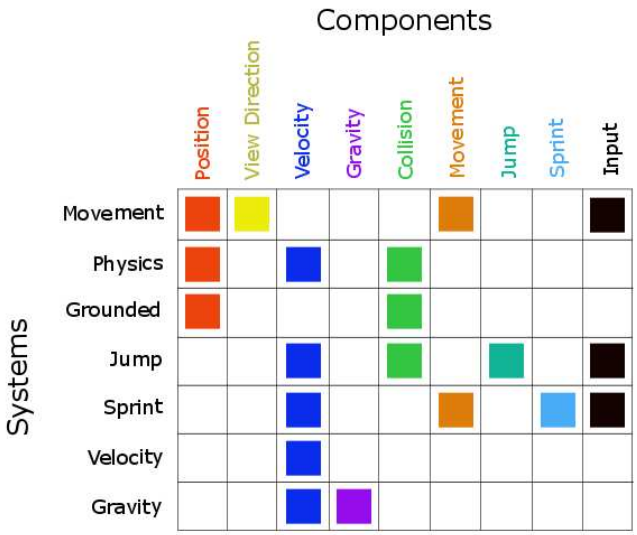
For our project, not only did we want to do as much as possible on our own, but we also knew that most of the open-source physics engines provide much more than what was needed. As one of the main reasons of this project was for us to learn as much as possible of what goes into making a voxel based game engine, implementing our own physics engine felt like the most obvious choice. Making a custom physics engine would also give us more control and would keep the complexity down to a minimum.

For our game engine to be as expandable as possible, being able to apply certain attributes to certain objects is very convenient. This is what makes an entity-component system, briefly explained in Section 1.5.2, very flexible and easy to change in contrast to an object oriented class hierarchy.

An entity-component system consists of entities and systems which handles these entities. An entity consists of different components, which holds different data. For example, a Health component would keep track of the health of an entity. The entity systems affects entities by modifying the data stored in their different components. Each system contains a list of the different entities that it affects. This list contains only the entities that are built up out of a certain set of components, which the system has specified.

There are different systems implemented that handle different parts of physics. Gravity is one of them. Any entity that is supposed to be affected by gravity is ensured to have a velocity and a gravity component. A gravity system is updated every frame and modifies the velocity component of every entity that has these two components.

Entity movement is handled in a similar fashion, with a movement-system making changes to every entity that has a movement component as well as a velocity component. The movement system changes the entities based on a few different parameters. One of these parameters is a third component, the input component, which can be affected by two different factors: player input or AI. This structure allows for very flexible use of the movement system as it is not bound to be used by the player only. When a field in the input component is changed, the movement system changes the velocity component of the same entity accordingly. An example of this would be if the player presses a key to move forward, the field for forward movement is changed by a controller. The movement system detects the changed field and increases the forward velocity of the player. This is the same approach as proposed by Boreal [66] whom the credit for this structure goes to. Figure 22 illustrates how entities, components and entity systems relates to each other.



Entity	Components	Affected by systems:
Player	All of the above	All of the above
Ball	Position, Velocity, Gravity, Collision	Physics, Grounded, Velocity, Gravity
Ghost	Position, View Direction, Velocity, Movement, Input	Movement, Velocity

Figure 22: Examples of systems affecting different entities depending on their components.

Movement Changes the velocity component of an entity depending on its input component.

Physics Changes the position of an entity based on its velocity component. Performs collision detection if the entity has a collision component.

Grounded Checks if the entity is in contact with the ground. Used in other systems such as the jump system.

Jump Changes the velocity component of an entity depending on its input component.

Sprint Temporarily increases the max velocity of the entity based on input.

Velocity Keeps the entities from exceeding the maximum velocity.

Gravity Changes the velocity of all entities with a gravity component.

4.2.2 Collision Detection

To make CD as cheap as possible, we are using AABBs as geometrical representation of blocks and other objects. The main reason for this is that AABBs are box shaped, which makes for perfect representation of the blocks. It also keeps the different calculations needed for CD to a minimum since, if different kinds of bounding volumes were used, different kinds of intersection tests would be required.

Getting into the specifics of CD, we decided early on that a sweep-and-prune algorithm or spatial partitioning would not be needed. With the game consisting of a huge 3D grid of different blocks, the world is already partitioned. The broad phase of our CD is done by creating a so called swept broad phase box, which is a box containing all positions a moving object could possibly end up at after a specific time frame. Data for each block inside this swept broad phase box is stored for later use in the narrow phase. The swept broad phase box is created with the following code section:

```
public static AABB getSweptBroadPhaseBox(AABB box, Vector3f v){

        float xMin, xMax, yMin, yMax, zMin, zMax;

        xMin = v.x > 0.0f ? box.min.x : box.min.x + v.x;
        yMin = v.y > 0.0f ? box.min.y : box.min.y + v.y;
        zMin = v.z > 0.0f ? box.min.z : box.min.z + v.z;
        xMax = v.x > 0.0f ? box.max.x + v.x : box.max.x;
        yMax = v.y > 0.0f ? box.max.y + v.y : box.max.y;
        zMax = v.z > 0.0f ? box.max.z + v.z : box.max.z;

        Vector3f min = new Vector3f(xMin, yMin, zMin);
        Vector3f max = new Vector3f(xMax, yMax, zMax);

        return new AABB(min, max);
}
```

`v` is the velocity vector of the moving object, `box.min` is the minimum point of the AABB that represents the moving object and `box.max` is the maximum point of the same AABB.

Following this is the narrow phase, which is when the CD calculations actually occur. The CD in our case is a swept AABB CD function. A common way of doing CD is to move an object, calculate intersection and, during the collision response, move the object again, if it is colliding. This can result in the following error; if an object is moving very fast or if the frame rate of the game is very low, the object could move through another object without the collision being detected. An example of this is illustrated in Figure 23.

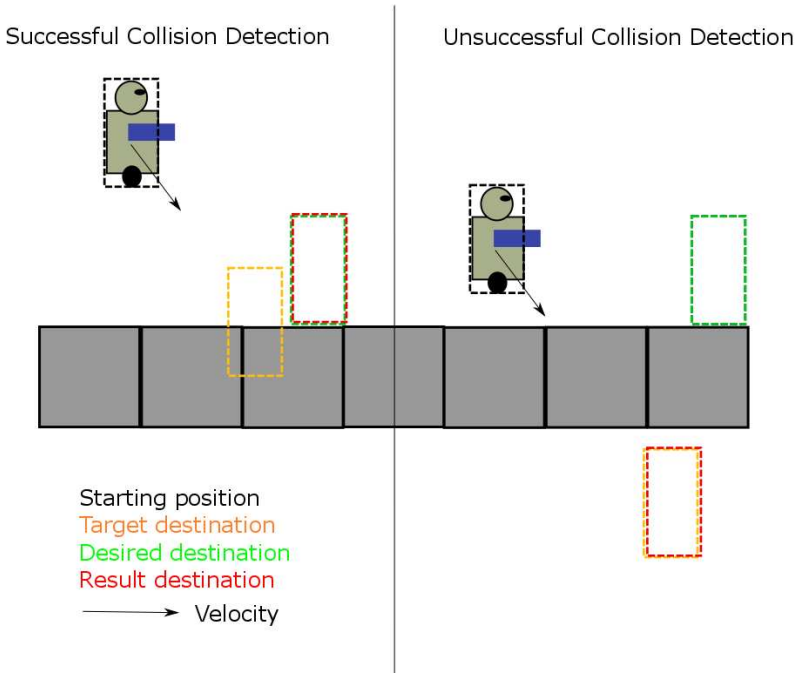


Figure 23: Image illustrating a successful and unsuccessful collision detection scenario

The swept AABB function prevents this by calculating if collision with another object is going to happen before the moving object is moved at all. The function calculates a resulting collision normal, in other words, a vector orthogonal to the plane that the object is colliding with. The function also calculates at what point in time during the frame the collision happens. This is done for every block that was stored during the broad phase in order to find out what block the object collides with first. The normal and time of the first collision are used in order to calculate where the object is going to end up and what velocity it is going to have. This process is repeated until no more collision happens during the specified time frame.

4.2.3 Block Selection

We chose to use a slightly modified version of the grid traversal algorithm, since the game world of this project is built using a fixed voxel grid. Not only did it utilize our data structures efficiently, we also found that it is well suited for a real-time scenario.

When allowing the player full freedom of movement you also want them to feel that they have full control over it, if the algorithm is inefficient or inaccurate it is more likely that it will detract from the experience instead of being a tool in increasing the immersiveness. Our implementation of this algorithm uses two floating point comparisons one floating point addition and a one integer addition, which achieves the end result of a fast and smooth target system.

The main disadvantage of this approach is that it does not handle dynamic objects. When making the data structure that encapsulates our voxels we ignored everything that is not blocks, which means that the player will not be able to target non-block entities such as NPCs.

4.3 Discussion

The CD in our engine has been tested extensively and has never resulted in the player getting stuck or falling through a block. The different choices of how to implement CD feels like they fit the game engine nicely.

However, there is a limitation to our solution, it cannot handle anything but regular voxels. This means that there is currently no way to implement stairs, since the algorithm only checks if a voxel is solid and in that case blocks the movement in that direction.

Another major shortcoming is the lack of collision between entities. Right now the system takes advantage of the built in data structures, which means that it cannot handle anything but static objects. Since no NPCs are generated in the world this is currently not a problem. However, in the future this would be an increasingly important feature to implement, as NPCs become a bigger part of the world.

The same situation applies to targeting. It uses a ray casting algorithm, which iterates through voxels in order to get the closest solid block. Entities on the other hand is not a part of the voxel data structure, which means that the ray will not take them into consideration. This is probably the most important component to add in the future, since features such as combat or interacting with entities cannot be implemented without it.

Since world interaction can differ a lot from game to game, implementing the different features was difficult. Even though we found a lot of different resources for learning the techniques needed, applying these techniques to our specific project was not always as easy as it seemed. While we think there is room for minor improvements on some parts, such as further optimizing the CD, we are satisfied with the overall result.

5
Artificial Intelligence

Artificial intelligence (AI) is a broad topic used in several areas such as stock trading and computer games [67, 68]. The term was first coined by John McCarthy in 1955 [69]. He described it as "the science and engineering of making intelligent machines" [70].

AI in a game can be used to simulate intelligence, most commonly to give the impression of intelligent behaviour to non-player characters (NPCs). Some of the common goals of AI is that the NPC is able to reason, plan, learn, move, manipulate the world, and interact with the player. In a game such as this, where the main feature is exploration, NPCs populating and wandering the world makes it feel more alive.

This chapter presents some options for implementing AI movement and AI behaviour, as well as motivating the choice of techniques and discussing the results of these choices.

5.1 Theory

This section covers the theory behind two fundamental aspects of AI, movement and behaviour.

5.1.1 AI Movement

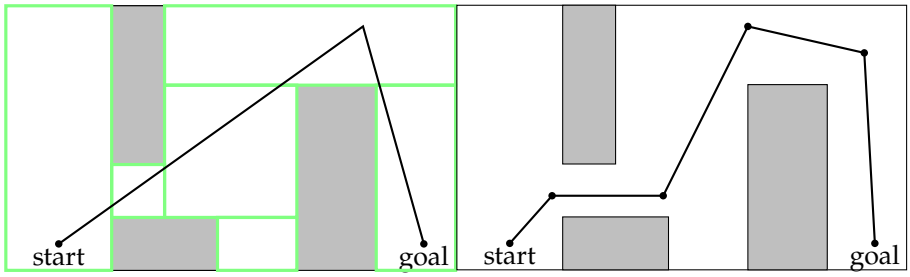
In many games the most important aspect of AI is the movement [71], as any intelligent decision-making is of no use if the NPC cannot move around intelligently. To solve the problem of AI movement, there are several techniques that can be used in order to simplify the game world representation. An algorithm can then be utilized to find a path on this representation.

The most common pathfinding algorithm used in games is A* [72] (pronounced as "A star"), a graph traversal algorithm. It is used to find the shortest path from a starting node to a goal node on a graph. This section will cover this algorithm as well as two of the most common methods that simplify the game world, navigation meshes [73] and waypoints [68]. These methods share a common factor, they aim to represent the game world as a graph data structure.

Game World Representation

A navigation mesh is a set of convex polygons that defines the traversable areas of an environment. Together these polygons form a graph, where each polygon is a node. Figure 24a shows one of many possible paths that can be taken over the polygons. The navigation mesh can either be calculated using algorithms or be manually created. In a dynamic world the mesh has to be updated each time the world is modified, making it computationally expensive. Hence, a navigation mesh is more suitable for static environments.

A waypoint system is a set of points, where each has a connection to one or more other points. In pathfinding, these points are treated as nodes, a connection between two nodes is a straight line as seen in Figure 24b. An NPC traversing a path in this system can only follow these lines,



(a) Navigation mesh. The green rectangles mark the navigation mesh.

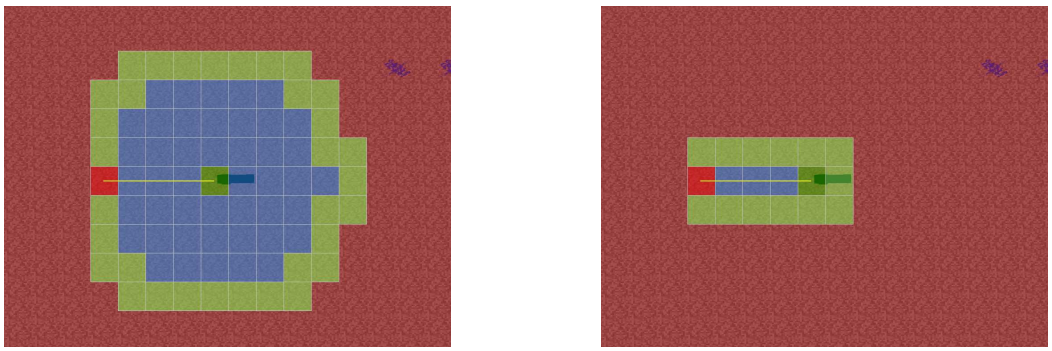
(b) Waypoints. the dots mark the waypoints.

Figure 24: Navigation mesh and waypoint system

creating less natural paths than those found using a navigation mesh. One of the advantages compared to navigation meshes is that it is simpler to create. Waypoints are more suitable for static environments since in a dynamic world the connections could easily be obstructed.

A* Pathfinding

The A* algorithm was first described in 1968 by Hart, et al [72]. It is an extension of Dijkstra's algorithm using a heuristic approach for better performance. This results in an algorithm that finds the path between two nodes on a graph with the lowest-cost, determined by using a cost function $f(x)$ for an arbitrary node x. The cost f, is the sum of two other functions, $g(x)$ and $h(x)$(the heuristic). The function h is an estimate of the distance from the current node x to the goal node. The function g is the already known distance from the starting to node to node x. Adding the heuristic function results in A* finding the shortest path while visiting less nodes than Dijkstra's algorithm, as seen in Figure 25.



(a) Dijkstra's algorithm, illustrating the often called 'flood-fill' result.

(b) A*, the algorithm avoids visiting several nodes due to the heuristic function.

Figure 25: Pathfinding from a NPC to the red node, blue nodes are on the closed list and green nodes are on the open list

The nodes in a graph that will be searched can be split up into three lists; a closed list containing all nodes that have been visited, an open list containing nodes to be visited, and nodes that have

not been examined. The open list is a priority queue in order to improve search complexity from $O(n)$ to $O(\log(n))$ [30], where the node with the lowest $f(x)$ has the highest priority.

A simplified breakdown of the algorithm is given here.

1. Let x be the starting node. Determine $f(x)$ and add it to the open list
2. Take a node A from the open list (the node with the lowest cost)
   (a) if A is the goal node, terminate, as a path has been found.
3. Add A to the closed list as it has been visited.
4. Let node B be a node connected to A, if B is not already on the closed or open list, determine $f(B)$ and add it to the open list.
5. Repeat step 4 for all connected nodes to A and then go back to Step 2.

To retrieve the actual path of nodes, the process can be modified so that a node B points to its parent A.

5.1.2 AI Behaviour

To add to the illusion of intelligence in the NPCs of a game, a behaviour model can be implemented. There are several approaches to modelling this behaviour, this section will cover two common ones, finite-state machine and fuzzy logic.

Finite-State Machines

According to Bourg and Seemann [68], one of the common ways to model the selection of behaviour is the usage of a finite-state machine [74]. The usage of these in games date back to to early computer games such as Pac Man, and are still popular today due to being easy to implement.

A finite-state machine consists of a finite number of states. Transitions between these states can be triggered by events or conditions that are met. When using this model, the decision making of the AI is determined by the state it is currently in. To use Pac-Man as an example [68], a ghost in the game has four states.

- Wander the maze
- Chase Pac-Man
- Return to Base
- Flee from Pac-Man

A transition in this example would be going from Wander the maze to Chase Pac-Man, this is triggered by an event, for example when a ghost spots Pac-Man.

Fuzzy Logic

Fuzzy logic is a term introduced by Lotfi A. Zadeh in 1966 [75]. It is an implementation where decisions are made depending on a number of non-binary inputs. That is, in a computer game the AI takes into consideration a number of variables, such as its own health and the players health, to determine if it should attack or not. This is in contrast to the finite-state machine, where

it is binary, either the AI is in an aggressive state, or it is not, and will attack the player based on this.

5.2 Result

This section describes the methods we utilized to implement AI movement and behaviour. These techniques were used for one NPC only, namely a rabbit.

5.2.1 AI Movement

The AI movement implemented in this game uses a modified version of A*, with the existing data structure representation of the world as the search area. This section covers our implementation of this and motivations for using it.

As discussed in Section 5.1.1, there are two common variations of representing the game world, a waypoint system or a navigation mesh. In our game, new parts of the world are generated as the player explores, and existing parts can be manipulated by the player as described in Chapter 4. This can result in the representation becoming obsolete, requiring a run-time recalculation. Although a mesh solution was not unthinkable, it was deemed unnecessary given the data structure we had available.

In our implementation, the search area for the pathfinding is a 3D grid of all the voxels currently loaded in memory, which is already stored by the engine. This grid is essentially a graph, where each voxel is a node. From a pathfinding perspective, all these voxels are not connected so that an NPC could move between them, since there are both solid and non-solid voxels. In order to use A*, one alternative is to calculate all connections in this grid that are actually valid for an NPC to walk through and use that as the search area instead. This is essentially a waypoint system and is unsuitable for reasons discussed in Section 5.1.1, but it has merits discussed later in this section. The other alternative, the one we chose, is to modify the A* algorithm to determine if connections are valid.

In our version of A*, we assume the NPC (rabbit) traversing the path is voxel-sized. It is affected by gravity, meaning it can not take paths where there is no surface to stand on. It can jump higher than the equivalent of one voxel. Consider the A* process as described in Section 5.1.1, in step 4 calculations are made to determine if an adjacent node can be traversed by the rabbit, adhering to the limitations described, before being added to the open list. The resulting path is a set of voxel coordinates that the rabbit can traverse.

This modification adds a constant increase in CPU usage, but does not increase time complexity in terms of big O notation. To avoid this increased usage, an alternative method that increases memory usage could be utilized by calculating the connections between voxels before-hand and storing this information in our grid. This would require calculations whenever the world is modified, increasing the cost as the number of dynamic factors increases, for example if NPCs were added which modify the world. We opted for the CPU intensive solution since this would scale better in a game with a dynamic world as key feature, and if more NPCs were added which follow different limitations, there would need to be be unique connections stored for each of them.

Unfortunately, A* can not determine if a path exists or not until all nodes have been evaluated, since the search area consists of all voxels currently loaded in memory, a large number of iterations can be needed to determine this. Our algorithm is therefore limited to a fixed amount of iterations

as a safe mechanism, that is, if a path can not be found sufficiently fast, no path is returned. To solve the problem of paths being compromised by the player, anti-stuck mechanisms have been implemented that causes the rabbit to recalculate its path. The distances that the rabbit attempts to travel with one path is also kept relatively short, as to reduce the cost of recalculation if a compromise happens.

5.2.2 AI Behaviour

Our implementation of AI behaviour is simplistic. Since the main focus of the project was to develop a game *engine*, with a focus on the technical aspect. There is no combat system or other gameplay elements in which the NPCs could interact with the player, due to these factors a finite-state model with three states was chosen.

Aggressive
: In the aggressive state, the rabbit attempts to walk towards the player

Evading
: In the evading state, the rabbit attempts to flee from the player

Passive
: In the passive state, the rabbit aimlessly wanders around and has no consideration for the players position relative to itself.

When there are not many different actions or decisions for the NPC to make, this simple state-machine is sufficient. However this implementation does not result in very exciting NPCs for the player, since there is no interaction. An additional feature that was considered but not implemented due to time constraints is for the rabbit to modify blocks in the world.

5.3 Discussion

There is a lot of room for improvement in the AI, primarily in behaviour and diversity of NPCs. With more time and gameplay elements, other NPCs than rabbits could be implemented, utilizing different behaviour models than the simple state-based one. For example, if a combat system were to be implemented, an aggressive NPC could be added that considers several factors such as its own health, the players health and the players weapons. Fuzzy logic would be a good choice in this case, where there are more variables to consider.

The voxel grid proved to be suitable for pathfinding, requiring only minor modifications of A* to be implemented, with no additional simplification of the game world required. The pathfinding has only been implemented for a rabbit which is voxel-sized. Creating new NPCs of larger sizes would test the flexibility of the pathfinding in the game engine, examining if a modified A* for our grid could create intelligent and efficient pathfinding for them as well.

6

Graphical User Interface

Graphical user interfaces (GUI) are used in order to allow users to interact with a computer in an effective and intuitive way. The purpose of GUIs is to provide an easy way to understand what the user can do and how to do it with as little effort as possible.

6.1 Theory

Two fundamental qualities of GUIs are effectiveness and how easy the GUI is to use. Effectiveness is a measurement in how fast a task can be performed and how many steps that are required, where steps are actions such as pressing a button or filling in a field. How easy a GUI is to use can be hard to determine, and is usually done by performing case studies and surveys [76, 77].

At times, the effectiveness of performing a task can be in conflict with how easy it is to understand how the task should be performed. For example, saving a file that the user works on could be done by either using a keyboard shortcut, or going through several menu steps which would take longer time. If the target user is not expected to have high experience with advanced GUIs, it is usually better to chose an easy to understand GUI, even when it sacrifices the effectiveness, and, if possible, also support the more effective way for advanced users. One way of designing a GUI is to create fictional personas that each represent the most important aspects of different user groups. These personas can then be used to evaluate how that user group would think about different aspects of the GUI [76, 77].

GUIs can be used in many ways to provide important functionality in games. Menus are generally used in order to let the user control some aspects of the game. In order to provide information about what is happening in the game, a heads up display (HUD) can be a suitable choice. HUDs are usually simple two dimensional graphics rendered as a last step of the current rendering frame. In Figure 26 a HUD from our game can be observed.

Figure 26: The HUD displays how much progress have been done in destroying a block and also what block is currently selected if the player decides to create a block.

6.2 Result

In our game engine, we wanted to let the user be able to customise the experience by changing various settings such as controls, audio, and graphics. In the controls menu, the user should be able change what key will be used to activate a certain action as well as adjusting the sensitivity of the mouse, that is, how much the player will rotate when the user moves the mouse. When creating a new world, we want to let the player specify a large amount of parameters that will control how the world is generated. For instance, some players might prefer to have more of some terrain types than others and can therefore chose this before generating a new world. For this we implemented a number of different menus. In this project, we decided to use a HUD to display what block type the player has selected and will be used when adding a block, how much time is left until a block has been removed, and a cross-hair which is in the center of the screen.

There exists a number of libraries for creating GUIs in LWJGL, such as Themable Widget Library (TWL) and Nifty GUI [78, 79]. We decided, however, to create our own GUI system, because we thought it would provide more flexibility and be easier to adjust to this project's needs.

We have used widgets as our main abstraction for GUI components. A widget's main purpose is to provide a way for the user to interact with the GUI. A button, for instance, can be used to let the user finish some setting configurations and return into the game. There are various specialised widgets for different purposes, but they all share a set of common operations: drawing itself, receive user input, determine if they should trigger an event, and also the possibility to be deactivated. In order to handle an event triggered by a widget, the user of that widget must implement an interface and register itself as a listener. When an event is triggered, the listener will be notified and can take appropriate action. In Figure 27, a graphical interface made with our GUI system can be observed.

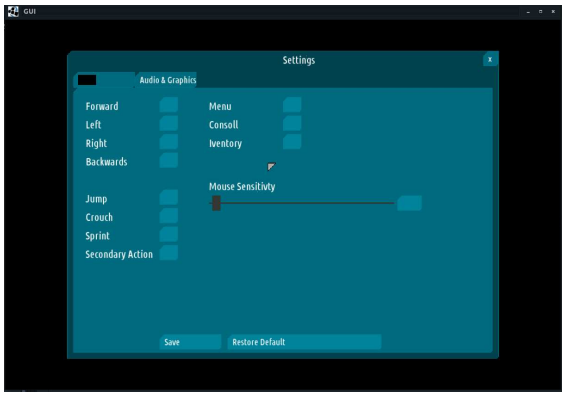


Figure 27: A settings view with tabs for changing panels, TextInput Fields for changing the key-bindings, a slider for changing the mouse sensitivity and close, save, and restore buttons.

One problem that exists when creating two dimensional graphics is how to handle different screen sizes. If no consideration is taken for this, the result might be that some parts will not be visible for users with unusual screen resolutions. We decided to use an approach where we specify the locations of the GUI elements in a virtual screen space. This virtual space will then be centered in the screen, hence, guaranteeing that everything that the designer of the GUI has made will be visible on all possible screens. In order to make this work, a matrix that will scale and translate the virtual space into the real screen space is needed. The scale will be the lowest

value required to scale either the virtual height equal to the screen height, or the virtual width equal to the screen width. For example, we could have a virtual screen with width 800 and height 600 and want to center this on a real screen with 1100 and 700 in width and height respectively. To scale the virtual width to the real width, we need to multiply with the real divided by the virtual, and same goes for the height.

```
widthScale = 1100/800 = 1.375
heightScale = 700/600 = 1.166..
```

Since the height scale is smaller, it will be used to scale the virtual screen. The last step is to ensure that the virtual screen is centered. This is done by using the following equation as a translation for the x coordinates. In Figure 28 the virtual screen is illustrated.

```
translation = (realWidth - (scale * virtualWidth)) / 2 = 86
```

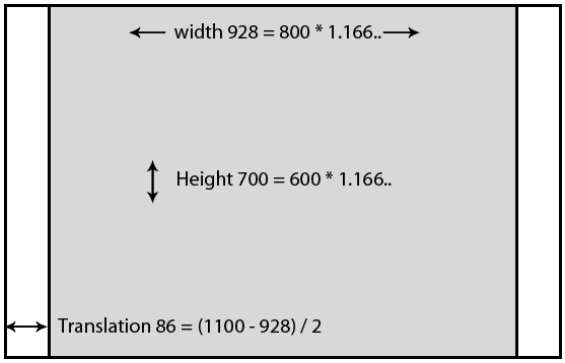


Figure 28: The grey represents the virtual screen centered into the real screen, which is white.

6.3 Discussion

In our game, we used this framework that we developed to make a start menu, an in-game menu, an HUD, and a terminal that can execute commands. Although implementing this framework required a large amount of work, we think it was worth the effort since we accomplished a simple way to create GUIs that are designed specially for our game engine.

7
Audio

To create an interesting game, as many of the human senses as possible need to be engaged. The first and most obvious way is visual feedback. However, acoustic feedback in form of audio effects or music is also of great importance in order to create an immersive gaming experience. For example, in many games, music might be used to set the mood and enforce a certain atmosphere. In an action game, fast-paced music can make a situation feel more intense. On the other hand, in the kind of game we have created, a more suitable choice would be calm and soothing music to help the player feel relaxed. Furthermore, Shorter sound effects can be used to signal to the player that something is happening that might not be obvious to the other senses [80].

7.1 Theory

To play back sound, a library is necessary for loading and manipulating audio. There are three candidates that we considered using; DirectX Audio, FMOD, and OpenAL. We evaluated these options based on several criteria that they needed to fulfill. First and most importantly, the library should be supported on as many platforms as possible and be free to use. Furthermore, we needed support for one or more common sound formats, such as WAV, MP3, or OGG.

7.1.1 DirectX Audio

The DirectX Audio library contains functionality for playback of both sound effects and music in a simple way for the programmer. One of its biggest disadvantages is its platform dependency, since it only works in a Microsoft environment. Also, there is no Java binding for DirectX Audio, which means we would have to create our own. This is out of the scope of our project [81, 82].

7.1.2 FMOD

FMOD is supported on a majority of the popular platforms, phones, video game consoles, and desktop computers. Currently, FMOD is being used in some of the most popular game engines such as CryEngine, Unity, and Unreal Engine [83]. However, FMOD is only free of charge if used in a non-commercial context. Even if we do not intend to distribute our game, we think this is an unnecessary limitation.

7.1.3 OpenAL

While OpenAL is not the most user friendly library and does not have a lot of features, it does support our basic needs. It is platform independent, supports playback of WAV, OGG, and even MP3 through some extensions. OpenAL is also completely free of charge, no matter the intended usage. The most appealing thing about OpenAL is the fact that its API is very similar to OpenGL. It also comes bundled with LWJGL, our framework of choice.

7.2 Result

Based on our comparisons above, we chose to to use OpenAL. We made this choice because it has all the features we needed for our voxel game engine and is supported on all our targeted platforms. Furthermore, it does not have any licensing fees, which enables us to freely distribute our engine.

Because OpenAL is a very basic API, we had no framework handling the loading of audio files and keeping track of them. We deemed it necessary to create some sort of wrapper for the OpenAL interface to keep track of the data and playback buffers for sounds. Moreover, our framework was designed to make it easy to load in an audio file and perform simple tasks, such as play, pause, rewind, and stop. We also wanted to be able to make modifications to sound properties, such as gain, pitch, and panning.

A data buffer is used to store the data of a sound that will be played. If we want two instances of the same sound to be played simultaneously, we want to utilize the same data buffer for the two instances. In order to do this, a separate playback buffer is used. The playback buffer keeps track of the playback progress and properties of the sound and holds a reference to the data buffer.

Since our game is three-dimensional, we wanted to utilize stereo sound that is standard for most systems. Using stereo speakers, a sound with a position in a three dimensional space can be simulated, meaning that, for example, a sound that is located to the left of the player will produce a louder sound in the left speaker. OpenAL is built around the concept of having one listener and multiple sound sources, as seen in Figure 29. The sources can be positioned relative to the listener and then the sound levels of the left and right speakers are calculated based on these positions.

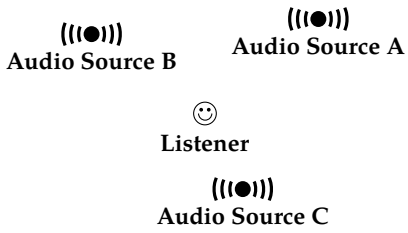


Figure 29: One listener and multiple sound sources.

In our audio framework, we have implemented an abstraction that divides the sound interface into two main parts. We have the listener, which has a position and a gain that acts as the master volume of all sounds. For the sound sources, there are two different types; regular and positional. The difference between the two is that the position of a positional sound will be taken into account during playback. It is also possible to set the velocity of a positional sound. This property can be used by OpenAL to calculate a Doppler effect of the sound [84]. The Doppler effect is when a sound is moving towards or away from the listener and the pitch of the sound changes relative to the velocity.

7.3 Discussion

Using OpenAL for our project went smoothly, and even though it lacks a lot of features compared to FMOD and DirectX audio, such as playback of compressed audio files, we never felt that OpenAL lacked any features that we needed. Even if OpenAL was quite complicated to use, writing a wrapper for it made it relatively easy to use in the engine. We managed to reduce the amount of code for loading and playing a sound from more than 20 lines down to two lines with our wrapper, as seen in the code snippet below.

```
ISound sound = Sound.createSound("adam_sjunger");
sound.play();
```

In conclusion, we are very happy with the result and that our solution is supported on all our target platforms. One minor complaint is that the homepage of OpenAL was under construction, and seemed to have been for a long time. At first we were unsure if it really was the official page.

8 Conclusion

This chapter concludes the report. It contains an explanation and discussion of the result of the project, as well as a discussion of what could have worked better.

8.1 Result

Our goal was to make a voxel game engine, and we have succeeded in implementing a basic engine, thus we would call this project a success. All areas in our original problem statement were researched and implemented, with some room for improvement.

A brief description of the end product is given below, see respective chapters for a more detailed description of the results.

Upon starting the game, the player is presented with a start menu, here they have the options of either creating a new world or loading a pre-existing one. On the world creation screen, there are several customizable parameters, such as terrain height, cave density, as well as available biomes. While in the game, the player can move around, and add or remove blocks. The type of block to be added can be selected from the user interface. A terminal can be used to issue commands, such as spawning rabbits and changing the time of day. As the player explores the world, it is continuously generated around them in order to give the illusion of an infinite world.

8.2 Discussion

During this project, we have learned a lot about game development related fields, such as computer graphics, procedural content generation, and physics engines.

We are satisfied with the resulting game engine given the time period we had to develop it. The game created works well as a technical demonstration of the voxel engine, but there are no real gameplay elements except for exploration and building. However, the focus was to implement the underlying engine, not to create an interesting game.

As mentioned in Section 1.4 aesthetic and gameplay aspects of the game were not a priority, the majority of time being spent on building the engine. Given more time we would likely have put more work into the visuals, as well as implementing additional gameplay elements.

There were several issues caused by less than optimal planning. One of those issues was a flawed usage of Scrum. In the initial stages we did not appoint an official Scrum master, and even after we did, no member had a solid understanding of Scrum. This lead to the project not following the Scrum doctrine properly. In hindsight, during the planning stage of the project the group should have decided who should be scrum master, and that person would spend time in the planning stage to learn the scrum methodology more in-depth. Another issue was a non-uniform vision of the program structure, resulting in time wasted on code refactoring during the intermediary state of the project. We believe that if a proper planning stage had been added to the initial

development period, where an overview of the program structure could be created, we could have prevented many of the aforementioned problems occurring during the project.

References

[1] Zachtronics Industries. *Infiniminer*. 2009.

[2] M. Persson. *Minecraft*. 2011. URL: https://minecraft.net/ (visited on 05/06/2015).

[3] J. Harris. *Game Design Essentials: 20 Open World Games*. 2015. URL: http://www.gamasutra.com/view/feature/1902/game_design_essentials_20_open_.php (visited on 05/15/2015).

[4] *Terasology*. 2015. URL: https://github.com/MovingBlocks/Terasology (visited on 05/14/2015).

[5] *Seed of Andromeda*. 2015. URL: https://www.seedofandromeda.com/ (visited on 05/14/2015).

[6] Hello Games. *No Man's Sky*. 2015. URL: http://www.no-mans-sky.com/ (visited on 05/14/2015).

[7] Scrum.org and Scruminc. *The Scrum Guide*. 2014. URL: www.scrumguides.org/scrum-guide.html (visited on 02/08/2015).

[8] KHRONOS GROUP. *OpenGL*. 2015. URL: https://www.opengl.org/ (visited on 05/15/2015).

[9] lwgjl.org. *LWJGL - Introduction*. 2015. URL: http://legacy.lwjgl.org/ (visited on 02/13/2015).

[10] S. Bilas. "A Data-Driven Game Object System". In: Game Developers Conference. Gas Powered Games. 2002. URL: http://gamedevs.org/uploads/data-driven-game-object-system.pdf.

[11] Github. *Minecraft*. 2011. URL: https://minecraft.net/ (visited on 06/01/2015).

[12] K. T. Claypool and M. Claypool. "On Frame Rate and Player Performance in First Person Shooter Games". In: *Multimedia Syst.* 13.1 (Sept. 2007), pp. 3–17.

[13] B. F. Janzen and R. J. Teather. "Is 60 FPS Better Than 30?: The Impact of Frame Rate and Latency on Moving Target Selection". In: *CHI '14 Extended Abstracts on Human Factors in Computing Systems*. CHI EA '14. ACM, 2014, pp. 1477–1482.

[14] Microsoft. *DirectX*. 2015. URL: https://msdn.microsoft.com/en-us/library/windows/desktop/bb205067\%28v=vs.85\%29.aspx (visited on 05/15/2015).

[15] S. Roettger et al. "Smart Hardware-accelerated Volume Rendering". In: *Proceedings of the Symposium on Data Visualisation 2003*. VISSYM '03. Eurographics Association, 2003, pp. 231–238.

[16] M. Levoy. "Display of Surfaces from Volume Data". In: *IEEE Comput. Graph. Appl.* 8.3 (May 1988), pp. 29–37.

[17] L. A. Westover. "Splatting: A Parallel, Feed-forward Volume Rendering Algorithm". UMI Order No. GAX92-08005. PhD thesis. Chapel Hill, NC, USA, 1991.

[18] P. Lacroute and M. Levoy. "Fast Volume Rendering Using a Shear-warp Factorization of the Viewing Transformation". In: *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '94. New York, NY, USA: ACM, 1994, pp. 451–458. URL: http://doi.acm.org/10.1145/192161.192283.

[19] R. Geiss. "Generating Complex Procedural Terrains Using the GPU". In: *GPU Gems 3*. Ed. by H. Nguyen. Addison-Wesley, 2008, pp. 7–37.

[20] D. Williams. "Volumetric Representation of Virtual Environments". In: *Game Engine Gems 1*. Ed. by E. Lengyel. Jones and Bartlett, 2010, pp. 39–60.

[21] F. E. Nicodemus et al. "Radiometry". In: ed. by L. B. Wolff et al. Jones and Bartlett Publishers, Inc., 1992. Chap. Geometrical Considerations and Nomenclature for Reflectance, pp. 94–145.

[22] F. C. Crow. "Shadow Algorithms for Computer Graphics". In: *SIGGRAPH Comput. Graph.* 11.2 (July 1977), pp. 242–248.

[23] L. Williams. "Casting Curved Shadows on Curved Surfaces". In: *SIGGRAPH Comput. Graph.* 12.3 (Aug. 1978), pp. 270–274.

[24] M. Knecht. *State of the Art Report on Ambient Occlusion*. Tech. rep. human contact: technical-report@cg.tuwien.ac.at. Institute of Computer Graphics and Algorithms, Vienna University of Technology, Nov. 2007.

[25] T. Akenine-Möller et al. *Real-Time Rendering 3rd Edition*. Natick, MA, USA: A. K. Peters, Ltd., 2008, p. 1045.

[26] P. Shanmugam and O. Arikan. "Hardware Accelerated Ambient Occlusion Techniques on GPUs". In: *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*. I3D '07. 2007, pp. 73–80.

[27] M. Lysenko. *Ambient occlusion for Minecraft-like worlds*. 2015. URL: http://0fps.net/2013/07/03/ambient-occlusion-for-minecraft-like-worlds/ (visited on 05/14/2015).

[28] B. Arnold. *Fast Flood Fill Lighting in a Blocky Voxel Game*. 2015. URL: https://www.seedofandromeda.com/blogs/29-fast-flood-fill-lighting-in-a-blocky-voxel-game-pt-1 (visited on 05/14/2015).

[29] S. Wolfram. "Statistical mechanics of cellular automata". In: *Reviews of Modern Physics* 55.3 (July 1983), pp. 601–644.

[30] T. H. Cormen et al. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009.

[31] M. Wloka. ""Batch, Batch, Batch:" What Does It Really Mean?" A talk given at Game Developers Conference. 2003.

[32] B. Mark and B. Tudor. "Procedural 3D Cave Generation". MA thesis. IT University of Copenhagen, 2014. URL: http://benjaminmark.dk/Procedural_3D_Cave_Generation.pdf.

[33] D. S. Ebert et al. *Texturing and Modeling: A Procedural Approach*. 3rd. Morgan Kaufmann Publishers Inc., 2002.

[34] M. Hendrikx et al. "Procedural Content Generation for Games: A Survey". In: *ACM Trans. Multimedia Comput. Commun. Appl.* 9.1 (Feb. 2013), 1:1–1:22.

[35] K. Perlin. "An Image Synthesizer". In: *SIGGRAPH Comput. Graph.* 19.3 (July 1985), pp. 287–296.

[36] J. B. Spjut et al. "Hardware-accelerated Gradient Noise for Graphics". In: *Proceedings of the 19th ACM Great Lakes Symposium on VLSI*. GLSVLSI '09. ACM, 2009, pp. 457–462.

[37] K. Perlin. "Improving Noise". In: *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '02. ACM, 2002, pp. 681–682.

[38] K. Perlin. *Standard for perlin noise*. US Patent 6,867,776. Mar. 2005. URL: http://www.google.com/patents/US6867776.

[39] K. Spencer. *OpenSimplexNoise.java*. 2015. URL: https://gist.github.com/KdotJPG/b1270127455a94ac5d19 (visited on 05/14/2015).

[40] S. Worley. "A Cellular Texture Basis Function". In: *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '96. ACM, 1996, pp. 291–294.

[41] K. Musgrave. *Procedural Fractal Terrains*. 2015. URL: https://www.classes.cs.uchicago.edu/archive/2014/winter/23700-1/project_4_and_5/MusgraveTerrain00.pdf (visited on 05/31/2015).

[42] L. Johnson et al. "Cellular Automata for Real-time Generation of Infinite Cave Levels". In: *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*. PCGames '10. ACM, 2010, 10:1–10:4.

[43] J. Cui et al. "A Voxel-based Octree Construction approach for Procedural Cave Generation". In: *International Journal of Computer Science and Network Security* (2011), pp. 160–168.

[44] Y. I. H. Parish and P. Müller. "Procedural Modeling of Cities". In: *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '01. ACM, 2001, pp. 301–308.

[45] G. Marsaglia. "Xorshift RNGs". In: *Journal of Statistical Software* 8.14 (July 2003), pp. 1–6. URL: http://www.jstatsoft.org/v08/i14.

[46] H. Walter and S.-W. Breckle. *Walter's Vegetation of the Earth*. Vol. 4. Berlin Heidelberg: Springer, 2002, pp. 527+.
[47] R. H. Whittaker. *Communities and ecosystems /*. 2d ed. Includes bibliographic references and index. Macmillan Publishing Co., c1975.
[48] A. Lugo et al. "The Holdridge life zones of the conterminous United States in relation to ecosystem mapping". In: (1999).
[49] R. Bailey. *Ecosystem Geography: From Ecoregions to Sites*. Statistics for Social and Behavioral Sciences. Springer, 2009.
[50] D. M. Olson et al. "Terrestrial Ecoregions of the World: A New Map of Life on Earth". In: *BioScience* 51 (2001), pp. 933–938.
[51] I. Horswill. "Lightweight Procedural Animation With Believable Physical Interactions". In: vol. 1. 1. Mar. 2009, pp. 39–49.
[52] Wikipedia. *Physics engine*. 2015. (Visited on 05/05/2015).
[53] *Bullet Physics Library*. URL: http://bulletphysics.org/wordpress/.
[54] M. T. Jones. *Open Source Physics Engines*. IBM. 2011. URL: http://www.ibm.com/developerworks/library/os-physicsengines/os-physicsengines-pdf.pdf (visited on 05/04/2015).
[55] J. Bender et al. "Interactive Simulation of Rigid Body Dynamics in Computer Graphics". In: *EUROGRAPHICS 2012 State of the Art Reports*. Eurographics Association, 2012.
[56] J. Nicholson. *The Concise Oxford Dictionary of Mathematics*. 5 ed. Oxford University Press, 2014.
[57] Team Meat. *Super Meat Boy*. 2010. URL: http://supermeatboy.com/ (visited on 05/29/2015).
[58] Valve Corporation. *Portal 2*. 2011. URL: http://www.thinkwithportals.com/ (visited on 05/29/2015).
[59] B. Kanber. *How Physics Engines Work*. Build New Games. Oct. 2012. URL: http://buildnewgames.com/gamephysics/ (visited on 06/05/2015).
[60] B. Mirtich. "Efficient Algorithms for Two-Phased Collision Detection". In: *Practical Motion Planning in Robotics: Current Approaches and Future Directions* (Dec. 1997). URL: http://www.merl.com/publications/docs/TR97-23.pdf. Massachusetts.
[61] A. Petersen. *Broad Phase Collision Detection Using Spatial Partitioning*. Build New Games. Oct. 8, 2012. URL: http://buildnewgames.com/broad-phase-collision-detection/ (visited on 05/05/2015).
[62] H. A. Sulaiman and A. Bade. "Bounding Volume Hierarchies for Collision Detection". In: *Computer Graphics*. 2012. URL: http://www.intechopen.com/books/computer-graphics/bounding-volume-hierarchies-for-collision-detection.
[63] C. Ericson. *Real-Time Collision Detection*. Focal Press, 2005.
[64] J. Amanatides and A. Woo. "A Fast Voxel Traversal Algorithm for Ray Tracing". In: *In Eurographics '87*. 1987, pp. 3–10.
[65] I. Millington. *Game Physics Engine Development: How to Build a Robust Commercial-Grade Physics Engine for your Game*. second edition. Taylor and Francis, 2010.
[66] Boreal. *Movement & Physics in an entity-component system*. May 30, 2013. URL: http://gamedev.stackexchange.com/questions/56519/movement-physics-in-an-entity-component-system (visited on 05/06/2015).
[67] T. C. Lin. "The New Investor". In: *UCLA Law Review* 60 (2013), pp. 680–734.
[68] D. M. Bourg and D. Seemann. *AI for Game Developers*. first edition. O'REILLY, 2004.
[69] J. McCarthy et al. *A Proposal for the Dartmouth Summer Research Project on Artificial Intelligence*. 1955.
[70] J. McCarthy. *WHAT IS ARTIFICIAL INTELLIGENCE?* Computer Science Department, Stanford University, Nov. 2007, p. 2. URL: http://www-formal.stanford.edu/jmc/whatisai.pdf.

[71] H. M. Ross Graham and S. Sheridan. "Pathfinding in Computer Games". In: *ITB Journal* 8.8 (2003), p. 57.
[72] P. E. Hart et al. "A formal basis for the heuristic determination of minimum cost paths". In: *IEEE Transactions on Systems, Science, and Cybernetics* SSC-4.2 (1968), pp. 100–107.
[73] van Toll W. et al. "Navigation meshes for realistic multi-layered environments". In: *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2011)*. IEEE, 2011, pp. 3526 –3532. URL: http://doi.acm.org/10.1145/566570.566636.
[74] G. H. Mealy. "A Method for Synthesizing Sequential Circuits". In: *Bell System Technical Journal* SSC-4.34 (1955), pp. 1045–1079.
[75] G. J. Klir and B. Yuan, eds. *Fuzzy Sets, Fuzzy Logic, and Fuzzy Systems: Selected Papers by Lotfi A. Zadeh*. River Edge, NJ, USA: World Scientific Publishing Co., Inc., 1996.
[76] T. Jenifer. *Designing Interfaces*. 2nd. O'REILLY & ASSOCIATES, 2011.
[77] A. Cooper et al. *About Face 3: The Essentials of Interaction Design*. John Wiley & Sons, Inc., 2007.
[78] l33tlabs. *TWL*. 2015. URL: http://twl.l33tlabs.org (visited on 05/13/2015).
[79] Nifty GUI. *Nifty GUI*. 2015. URL: http://void256.github.io/nifty-gui/ (visited on 05/13/2015).
[80] R. Stevens and D. Raybould. *The Game Audio Tutorial: A Practical Guide to Sound and Music for Interactive Games*. Focal Press, 2011.
[81] I. Munoz. *Building a Drum Machine with DirectSound*. URL: https://msdn.microsoft.com/en-us/library/ms973091.aspx.
[82] J. Boer. *Game Audio Programming*. Charles River Media, 2002.
[83] fmod.org. *FMOD - Engine Partners*. 2015. URL: http://www.fmod.org/ (visited on 05/06/2015).
[84] G. Nayler. *Dictionary of mechanical engineering*. DICTIONARY OF MECHANICAL ENGINEERING. Society of Automotive Engineers, 1996. URL: https://books.google.se/books?id=qfRSAAAAMAAJ.